Data Science at SAMY

Table of contents

1 Welcome to the Data Science Handbook

Welcome to our Data Science Handbook, your comprehensive guide to the methods, case studies, and best practices that define our approach to data science here at SAMY. This handbook is designed to be a dynamic resource for our team, evolving with new insights, tools, and technologies.

Within these pages, you'll find detailed case studies showcasing our successful projects, high-level concepts that underpin our strategies, and practical coding examples to help you apply these techniques in your own work. Irrespective of your experience in data science, this handbook aims to provide valuable insights and practical guidance to enhance your skills and knowledge.

We believe that sharing knowledge and continuously learning are key to staying ahead in the fast-paced world of data science. As such, this handbook is not just a static document but a collaborative space where ideas are exchanged, and innovation thrives.

Happy data sciencing!

> **Note**
>
> If you have any questions at all, ask any member of the team. Whilst this Handbook aims to be a valuable resource for self-learning, it can often be more beneficial to spend 5 minutes talking through a concept with someone on the team who may be able to describe something in a different manner to this document.

Part I

Introduction

2 The Data Science team

2.1 Where we sit

The Data Science department are a fully global resource within the alliance

Capture Intelligence is our biggest internal "client" as there are plenty of opportunities to offer data science led services in the research offering of Capture. But also have our own core central pipe for development that supports *all* agency brands.

What this means is as a team we have responsibilities that range from continual development of our own tech stack to help answer specific research questions for external clients to helping empower members of the alliance to use mindful applications of emerging technologies.

2.2 Who we are

Mike Tapp: Data Director

Jack Penzer: Global Data Product Lead

Jamie Hudson: Senior Data Scientist

Aoife Ryan: Data Scientist

Sophie Thomas: Jr. Data Scientist

Part II

Case Studies

3 Peaks and Pits

"Peaks and Pits" is one of our fundamental project offerings and a workflow that is a solid representation of good data science work that we perform.

3.1 What is the concept/project background?

Strong memories associated to brands or products go deeper than simple positive or negative sentiment. Most of our experiences are not encoded in memory, rather what we remember about experiences are changes, significant moments, and endings. In their book "The Power of Moments", two psychologists (Chip and Dan Heath) define these core memories as Peak and Pits, impactful experiences in our lives.

Broadly, peak moments are experiences that stand our memorable in our lives in a positive sense, whereas pit moments are impactful negative experiences.

Microsoft tasked us with finding a way to identify these moments in social data- going beyond 'simple' positive and negative sentiment which does not tell the full story of consumer/user experience. The end goal is that by providing Microsoft with these peak and pit moments in the customer experience, they can design peak moments in addition to simply removing pit moments.

3.1.1 The end goal

With these projects the core final 'product' is a collection of different peaks and pits, with suitable representative verbatims and an explanation to understand the high-level intricacies of these different emotional moments.

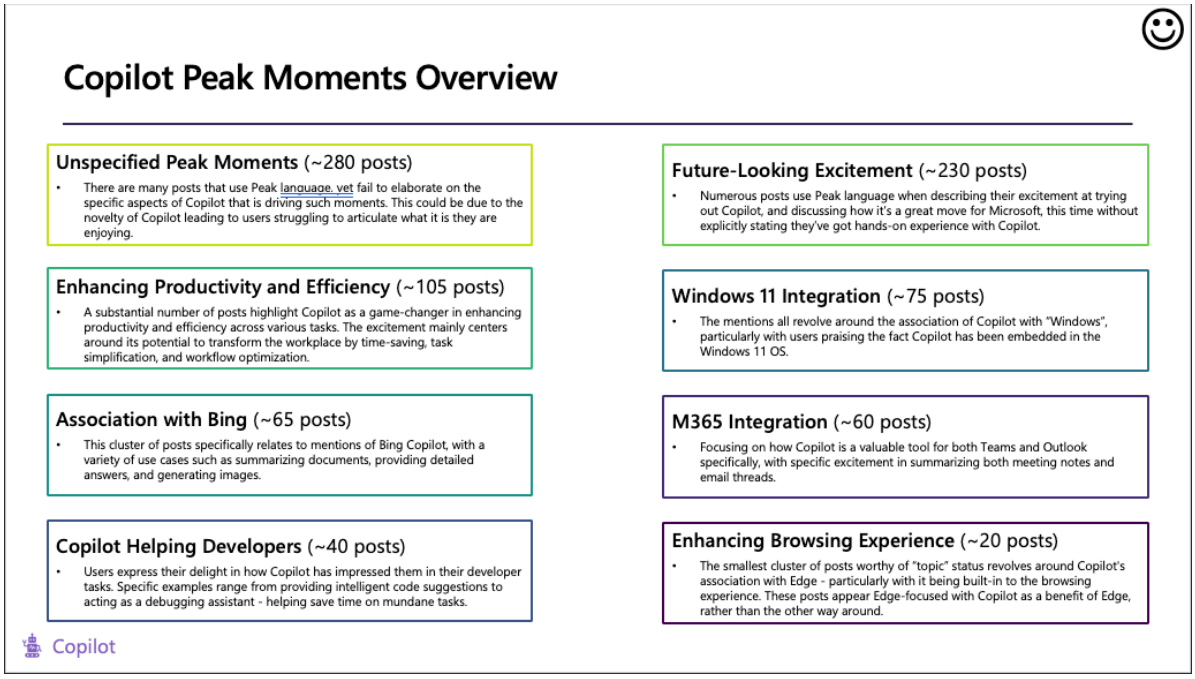


Figure 3.1: Screenshot from a Peaks and Pits project showcasing the identified Peak moments for a product at a high level

3.1.2 Key features of project

- There is no out-of-the-box ML model available whose purpose is to classify social media posts as either peaks or pits (i.e. we cannot use a ready-made solution, we must design our own bespoke solution).
- There is limited data available
  - Unlike the case of spam/ham or sentiment classification, there is not a bank of pre-labelled data available for us to leverage for 'traditional ML'.
- Despite these issues, the research problem itself is well defined (**what** are the core peak and pit moments for a brand/product), and because there are only three classes (peak, pit, or neither) which are based on extensive research, the classes themselves are well described (even if it is case of "you know a peak moment when you see it").

3.2 Overview of approach

Peaks and pits projects have gone through many iterations throughout the past year and a half. Currently, the general workflow is to use utilise a model framework known as SetFit to

efficiently train a text classification model with limited training data. This fine-tuned model is then able to run inference over large datasets to label posts as either peaks, pits, or neither. We then utilise the LLM capabilities to refine these peak and pit moments into a collection of posts we are extremely confident are peaks and/or pits. We then employ topic modelling to identify groups of similar peaks and pits to help us organise and discover hidden topics or themes within this collection of core moments.

This whole process can be split into six distinct steps:

1. Extract brand/product mentions from Sprinklr (the start of any project)
2. Obtain project-specific exemplar posts to help fine-tune a text classification model
3. Perform model fine-tuning through contrastive learning
4. Run inference over all of the project specific data
5. Use GPT-3.5 for an extra layer of classification on identified peaks and pits
6. Turn moments into something interpretable using topic modelling

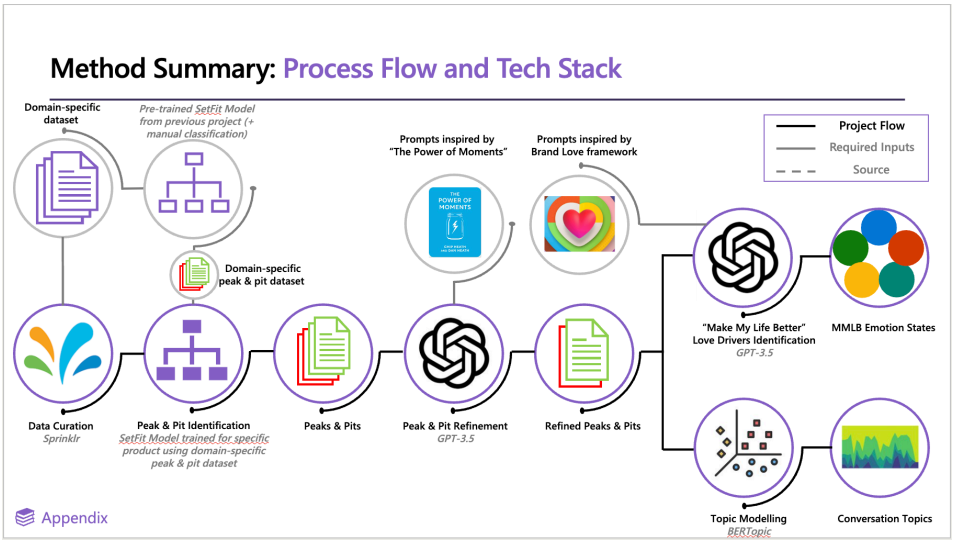


Figure 3.2: Schematic workflow from Project 706 - Peaks and Pits in M365 Apps

3.2.1 Obtain posts for the project (Step 1)

This step relies on the analysts to export relevant mentions from Sprinklr (one of the social listening tools that analysts utilise to obtain social data), and therefore is not detailed much here. What is required is one dataset for each of the brands/products, so they can be analysed separately.

3.2.2 Identify project-specific exemplar peaks and pits to fine-tune our ML model (Step 2)

This step is synonymous with data labelling required for any machine learning project where annotated data is not already available.

Whilst there is no one-size-fits-all for determining the amount of data required to train a machine learning model, for traditional models and tasks, the number of examples per label is often in the region of thousands, and often this isn't even enough for more complex problems.

The time required to accurately label thousands of peaks and pits to train a classification model in the traditional way is sadly beyond the scope of feasibility for our projects. As such we needed an approach that did not rely on copious amounts of pre-labelled data.

This is where SetFit comes in. As mentioned previously, SetFit is a framework that enables us to efficiently train a text classification model with limited training data.

> How does it do this?
>
> *Note the below is directly copied from the SetFit documentation. It is so succinctly written that trying to rewrite it would not do it justice.*
> Every SetFit model consists of two parts: a **sentence transformer** embedding model (the body) and a **classifier** (the head). These two parts are trained in two separate phases: the **embedding finetuning phase** and the **classifier training phase**. This conceptual guide will elaborate on the intuition between these phases, and why SetFit works so well.
> **Embedding finetuning phase**
> The first phase has one primary goal: finetune a sentence transformer embedding model to produce useful embeddings for our classification task. The Hugging Face Hub already has thousands of sentence transformer available, many of which have been trained to very accurately group the embeddings of texts with similar semantic meaning.
> However, models that are good at Semantic Textual Similarity (STS) are not necessarily immediately good at our classification task. For example, according to an embedding model, the sentence of 1) "`He biked to work.`" will be much more similar to 2) "`He drove his car to work.`" than to 3) "`Peter decided to take the bicycle to the beach party!`". But if our classification task involves classifying texts into transportation modes, then we want our embedding model to place sentences 1 and 3 closely together, and 2 further away.
> To do so, we can finetune the chosen sentence transformer embedding model. The goal here is to nudge the model to use its pretrained knowledge in a different way that better aligns with our classification task, rather than making the completely forget what it has learned.
> For finetuning, SetFit uses **contrastive learning**. This training approach involves

creating **positive and negative pairs** of sentences. A sentence pair will be positive if both of the sentences are of the same class, and negative otherwise. For example, in the case of binary "positive"-"negative" sentiment analysis, ("`The movie was awesome`", "`I loved it`") is a positive pair, and ("`The movie was awesome`", "`It was quite disappointing`") is a negative pair.
During training, the embedding model receives these pairs, and will convert the sentences to embeddings. If the pair is positive, then it will pull on the model weights such that the text embeddings will be more similar, and vice versa for a negative pair. Through this approach, sentences with the same label will be embedded more similarly, and sentences with different labels less similarly.
Conveniently, this contrastive learning works with pairs rather than individual samples, and we can create plenty of unique pairs from just a few samples. For example, given 8 positive sentences and 8 negative sentences, we can create 28 positive pairs and 64 negative pairs for 92 unique training pairs. This grows exponentially to the number of sentences and classes, and that is why SetFit can train with just a few examples and still correctly finetune the sentence transformer embedding model. However, we should still be wary of overfitting.
Classifier training phase
Once the sentence transformer embedding model has been finetuned for our task at hand, we can start training the classifier. This phase has one primary goal: create a good mapping from the sentence transformer embeddings to the classes.
Unlike with the first phase, training the classifier is done from scratch and using the labelled samples directly, rather than using pairs. By default, the classifier is a simple **logistic regression** classifier from scikit-learn. First, all training sentences are fed through the now-finetuned sentence transformer embedding model, and then the sentence embeddings and labels are used to fit the logistic regression classifier. The result is a strong and efficient classifier.
Using these two parts, SetFit models are efficient, performant and easy to train, even on CPU-only devices.

There is no perfect number of labelled examples to find per class (i.e. peak, pit, or neither). Whilst in general more exemplars (and hence more training data) is beneficial, having fewer but high quality labelled posts is far superior than more posts of poorer quality. This is extremely important due to the contrastive nature of SetFit where it's superpower is making the most of few, extremely good, labelled data.

Okay so now we know why we need labelled data, and we know what the model will do with it, *what is our approach* for obtaining the labelled data?

Broadly, we use our human judgement to read a post from the current project dataset, and manually label whether we think it is a peak, a pit, or neither. To avoid us just blindly reading through random posts in the dataset in the hope of finding good examples (this is not a good use of time), we can employ a couple of tricks to narrow our search region to posts that have

a reasonable likelihood of being suitable examples.

1) The first trick is to use the OpenAI API to access a GPT model. This involves taking a sample of posts (say ~1000) and running these through a GPT model, with a prompt that asks the model to classify each post into either a peak, pit, or neither. This is possible because GPT models have learned knowledge of peaks and pits from its training on large datasets. We can therefore get a human to only sense-check/label posts that GPT also believes are peaks or pits.
2) The second trick involves using a previously created SetFit model (i.e. from an older project), and running inference over a similar sample of posts (again, say ~1000).

We would tend to suggest the OpenAI route, as it is simpler to implement (in our opinion), and often the old SetFit model has been finetuned on data from a different domain so it might struggle to understand domain specific language in the current dataset. However, be aware if it not as scalable as using an old SetFit model and has the drawback of being a prompt based classification of a black-box model (as well as issues relating to cost and API stability).

Irrespective of which approach is taken, by the end of this step we need to have a list of example posts we are confident represent what a peak or pit moment looks like for each particular product we are researching, including posts that are "neither".

> **i** Why do we do this for each project?
>
> After so many projects now don't we already have a good idea of what a peak and pit moment for the purposes of model training?
> Each peak and pit project we work on has the potential to introduce 'domain' specific language, which a machine learning classifier (model) may not have seen before. By manually identifying exemplar peaks and pits that are project-specific, this gives our model the best chance to identify emotional moments appropriate to the project/data at hand.
> The obvious case for this is with gaming specific language, where terms that don't necessarily relate to an 'obvious' peak or pit moment could refer to one the gaming conversation, for example the terms/phrases "GG", "camping", "scrub", and "goat" all have very specific meanings in this domain that differ from their use in everyday language.

3.2.3 Train our model using our labelled examples (Step 3)

Before we begin training our SetFit model with our data, it's necessary to clean and wrangle the fine-tuning datasets. Specifically, we need to mask any mentions of brands or products to prevent bias. For instance, if a certain brand frequently appears in the training data within peak contexts, the model could erroneously link peak moments to that brand rather than learning the peak-language expressed in the text.

This precaution should extend to all aspects of our training data that might introduce biases. For example, as we now have examples from various projects, an overrepresentation of data from 'gaming projects' in our 'peak' posts within our training set (as opposed to the 'pit' posts) could skew the model into associating gaming-related language more with peaks than pits.

Broadly the cleaning steps that should be applied to our data for finetuning are:

- Mask brand/product mentions
- Remove hashtags #
- Remove mentions
- Remove URLs
- Remove emojis

> **What about other cleaning steps?**
>
> You will notice here we do not remove stop words-. As explained in the previous step, part of the finetuning process is to finetune a sentence embedding model, and we want to keep stop words so that we can use the full context of the post in order to finetune accurate embeddings.

At this step, we can split out our data into training, testing, and validation datasets. A good rule of thumb is to split the data 70% to training data, 15% to testing data, and 15% to validation data. By default, SetFit oversamples the minimum class within the training data, so we *shouldn't* have to worry too much about imbalanced datasets- though be aware if we have extreme imbalanced we will end up sampling the same contrastive pairs (normally positive pairs) multiple times. However, our experimentation has shown that class imbalance doesn't seem to have a significant effect to the training/output of the SetFit model for peaks and pits.

We are now at the stage where we can actually fine-tune the model. There are many different parameters we can change when fine-tuning the model, such as the specific embedding model used, the number of epochs to train for, the number of contrastive pairs of sentences to train on etc. For more details, please refer to the Peaks and Pits Playbook

We can assess model performance on the testing dataset by looking at accuracy, precision, recall, and F1 scores. For peaks and pits, the most important metric is actually **recall** because in step 4 we reclassify posts using GPT, so we want to make sure we are able to provide *as many true peak/pit moments as possible* to this step, even if it means we also provide a few false positives.

Click here for more info as to why recall is most important

As a refresher, `precision` is the *proportion of positive identifications* that are actually *correct* (it focuses on the correctness of positive predictions) whereas `recall` is the *proportion of actual positives* that are identified correctly (it focuses on capturing all relevant instances).
In cases where false positives need to be minimised (incorrectly predicting a non-event as an event) we need to prioritise `precision` - if you've built a model to identify hot dogs from regular ol' doggos, high precision ensures that normal dogs are not misclassified as hot dogs.
In cases where false negatives need to be minimised (failing to detect an actual event) we need to prioritise `recall` - in medical diagnoses we need to minimise the number of times a patient is incorrectly told they *do not* have a disease when in reality they *do* (or worded differently, we need to ensure that ***all*** patients with a disease are identified).
To apply this to our problem- we want to be sure that we capture all (or as many as possible) relevant instances of peaks or pits- even if a few false positives come in (neither posts that are incorrectly classified as peaks or pits). As we use GPT to make further peak/pit identifications, it's better to provide GPT with with a comprehensive set of potential peaks and pits, including some incorrect ones, than to miss out on critical data.

Visualise model separation

As a bonus, we can actually neatly visualise how well our finetuning of the sentence transformer embedding model has gone- by seeing how well the model is able to separate our different classes in embedding space.

We can do this by visualising the 2-D structure of the embeddings and see how they cluster:

This is what it looks the embeddings space looks like on an un-finetuned model:

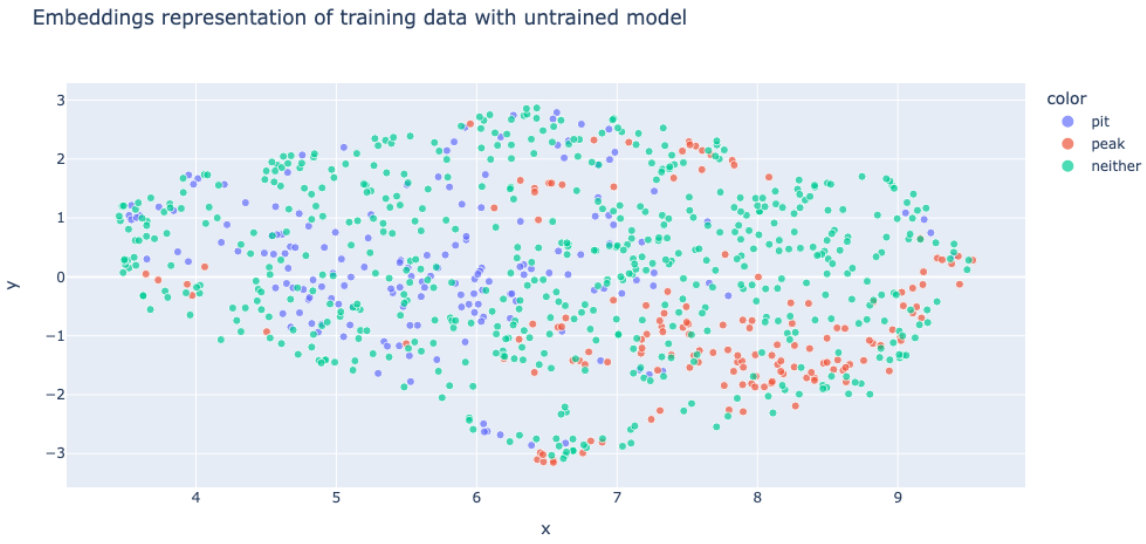


Figure 3.3: Un-finetuned embedding model

Here we can see that posts we know are peaks, pits, and neither all overlap and there is no real structure in the data. Any clustering of points observed are probably due to the posts' semantic similarity (c.f. the mode of transport example above). We would not be able to nicely use a classifier model to get a good mapping from this embedding space to our classes (i.e. we couldn't easily separate classes here).

By visualising the same posts after finetuning the embedding model, we get something more like this, where we can see that the embedding model now clearly separates posts based on their peak/pit classification (though we must be wary of overfitting!).

Figure 3.4: Finetuned embedding model

Finally, now we are happy with our model performance based on the training and validation datasets, we can evaluate the performance of this final model using our testing data. This is data that the model has never seen, and we are hoping that the accuracy and performance is similar to that of the validation data. This is Machine Learning 101 and if a refresher is needed for this there are plenty of resources online looking at the role of training, validation, and testing data.

3.2.4 Run inference over project data (Step 4)

It is finally time to infer whether the project data contain peaks or pits by using our fine-tuned SetFit model to classify the posts.

Before doing this again we need to make sure we do some data cleaning on the project specific data.

Broadly, this needs to match the high-level cleaning we did during fine-tuning stage:

- Mask brand/product mentions (using RoBERTa-based model [or similar] and `Rivendell` functions)
- Remove hashtags #
- Remove mentions
- Remove URLs
- Remove emojis

⚠ Note on social media sources

Currently all peak and pit projects have been done on Twitter or Reddit data, but if a project includes web/forum data quirky special characters, numbered usernames, structured quotes etc. should also be removed.

Okay now we can *finally* run inference. This is extremely simple and only requires a couple of lines of code (again see the Peaks and Pits Playbook for code implementation)

3.2.5 The metal detector, GPT-3.5 (Step 5)

During step 4 we obtained peak and pit classification using few-shot classification with SetFit. The benefit of this approach (as outlined previously) is its speed and ability to classify with very few labelled samples due to contrastive learning.

However, during our iterations of peak and pit projects, we've realised that this step still classifies a fair amount of non-peak and pit posts incorrectly. This can cause noise in the downstream analyses and be very time consuming for us to further trudge through verbatims.

As such, the aim of this step is to further our confidence in our final list of peaks and pits to be *actually* peaks and pits. Remember before we explained that for SetFit, we focussed on **recall** being the most important measure in our business case? This is where we assume that GPT-3.5 enables us to remove the false positives due to it's incredibly high performance.

❗ Why not use GPT from the start?

Using GPT-3.5 for inference, even over relatively few posts as in peaks and pits, is expensive both in terms of time and money. Preliminary tests have suggested it is in the order of magnitude of thousands of times slower than SetFit. It is for these reasons why we do not use GPT-x models from the get go, despite it's obvious incredible understanding of natural language.

Whilst prompt-based classification such as those with GPT-3.5 certainly has its drawbacks (dependency on prompt quality, prompt injections in posts, handling and version control of complex prompts, unexpected updates to the model weights rendering prompts ineffective), the benefits include increased flexibility in what we can ask the model to do. As such, in the absence of an accurate, cheap, and quick model to perform span detection, we have found that often posts identified as peaks/pits did indeed use peak/pit language, but the context of the moment was not related to the brand/product at the core of the research project.

For example, take the post that we identified in the project 706, looking for peaks and pits relating to PowerPoint:

> This brings me so much happiness! Being a non-binary graduate student in STEM academia can be challenging at times. Despite using my they/them pronouns during introductions, emails, powerpoint presentations, name tags, etc. my identity is continuously mistaken. Community is key!

This is clearly a 'peak', however it is not accurate or valid to attribute this memorable moment to PowerPoint. Indeed, PowerPoint is merely mentioned in the post, but is not a core driver of the Peak which relates to feeling connection and being part of a community. This is as much a PowerPoint Peak as it is a Peak for the use of emails.

Therefore, we can engineer our prompt to include a caveat to say that the specific peak or pit moment must relate directly to the brand/product usage (if relevant).

3.2.6 Topic modelling to make sense of our data (Step 6)

Now we have an extremely refined set of posts classified as either peak or pits. The next step is to identify what these moments actually relate to (i.e. identify the topics of these moments through statistical methods).

To do this, we employ topic modelling via BERTopic to identifying high-level topics that emerge within the peak and pit conversation. This is done separately for each product and peak/pit dataset (i.e. there will be one BERTopic model for product A peaks, another BERTopic model for product A pits, an additional BERTopic model for product B peaks etc).

We implement BERTopic using the R package BertopicR. As there is already good documentation on BertopicR this section will not go into any technical detail in regards to implementation.

From BertopicR. we end up with a topic label for each post in our dataset, meaning we can easily quantify the size of each topics and visualise temporal patterns of topic volume etc.

4 Conversation Landscape

The 'Conversation Landscape' method has proven to be an effective tool for querying, auditing, and analysing both broad concepts and finely grained topics across social conversations on all major platforms, as well as web pages and forums.

4.1 Project Background

Working with semi-structured or unstructured high-dimensional data, such as text (and in our case, social media posts), poses significant challenges in measuring or quantifying the language used to describe any specific phenomena. One common approach to quantifying language is topic modelling, where a corpus (or collection of documents) is processed and later represented in neater and simplified format. This often involves displaying top terms, verbatims, or threads highlighting any nuances or differences within the data. Traditional topic modelling or text analysis methods, such as Latent Dirichlet Allocation (LDA), operate on the probability or likelihood of terms or n-grams belonging to a set number of topics.

The Conversation Landscape workflow offers a slightly different solution and one that partitions text data without a specific need for burdening the user with sifting through rows of data in order to segment documents with hopes of understanding or recognising any differences in language, which would ideally be defined more simply as topics. The is mostly achieved through sentence transforming, where documents are converted from words to numerical values, which are often referred to as 'embeddings'. These values are calculated based on their content's semantic and syntactic properties. The transformed values are then processed again using dimension reduction techniques, making the data more suitable for visualisation. Typically, this involves reducing to two dimensions, though three dimensions may be used to introduce another layer of abstraction between our data points. The example provided throughout this chapter, represents some text data as nodes upon a two-dimensional space.

> **Note**
>
> *This documentation will delve deeper into the core concepts of sentence transforming and dimension reduction, along with the different methods used to cluster or group topics once the overall landscape is mapped out, referring back to our illustrated real-world business use case of these techniques. We will then later look at best practices and any downstream flourishes that will help us operate within this work-stream.*

4.2 Final output of project

An ideal output, like the one shown below should always showcase the positioning of our reduced data points onto the semantic space, along with any topic or subtopic explanations alongside, using color coding where appropriate. While we sometimes provide raw counts of documents per topic/subtopic, we always include the percentage of topic distribution across our data, occasionally referred to as Share of Voice (SOV).

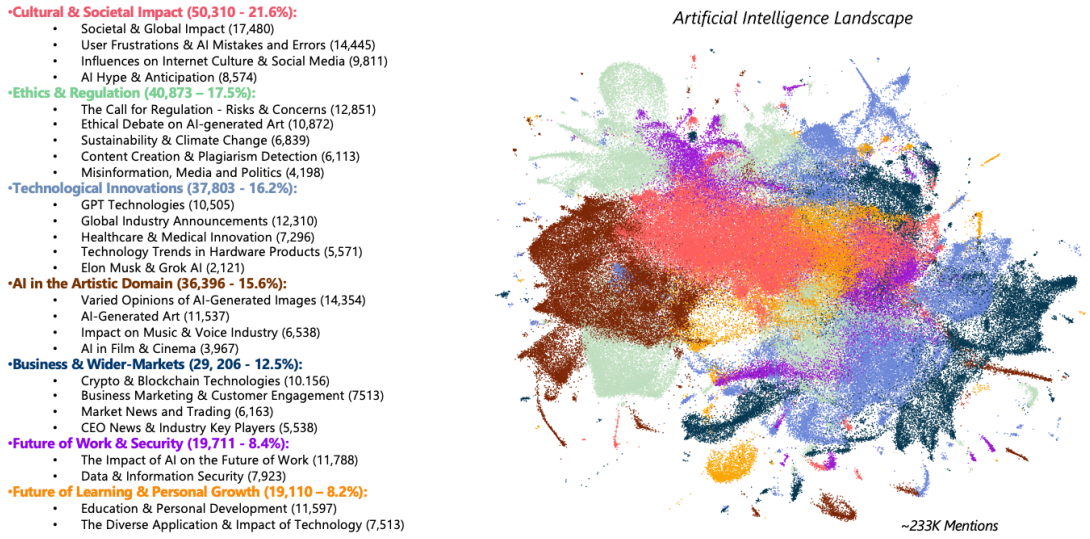


Figure 4.1: Screenshot Taken from the Final Output of an AI Landscape Microsoft Project - Q2 FY24

4.3 How to get there

As promised, we will provide some more context as well as the appropriate information surrounding the required steps taken, so that a reader may replicate and implement the methods mentioned throughout so far, providing an efficient analysis tool to use for any set of documents, regardless of domain specifics. While the example output provided displays a simplified means for visualising complex and multifaceted noisy data such as the 'Artificial Intelligence' conversation on social, there are a number of steps that one must take carefully and be mindful of throughout, in order to create the best fit model appropriate for a typical Conversation Landscape project.

The broad steps would include, and as one might find across many projects within the realms of Natural Language Processing (NLP):

- Initial Exploratory Data Analysis (EDA): Checking that the data is relevant and fit to answer the brief.
- Cleaning and Processing: Removal of spam, unhelpful or irrelevant data, and pre-processing of text variable for embedding.
- Transforming/Embedding: Turning our words into numbers which will later be transformed again before being visualised.
- Dimension Reduction: Reducing our representational values of documents to a manageable state in order to visualise.
- Topic Modelling/Clustering: Scientifically modelling and defining our data into a more digestible format.

4.3.1 Exploratory Data Analysis (EDA):

Whether the user is responsible for data querying/collection or not, the first steps in our workflow should always involve some high-level checks before we proceed with any of the following steps in order to save time downstream and give us confidence to carry over into the data cleaning and processing steps and beyond.

First, one should always check things like the existing variables and clean or rename any where necessary. This step requires a little forward thinking as to what columns are necessary to complete each stage of the project. Once happy with our variables, we can then check for things such as missing dates, and/or if there are any abnormal distributions across columns like 'Social Platform' that might skew any findings or help us understand or perhaps justify the resulting topic model. Next, we can do some bespoke or project specific checks like searching for likely-to-find terms or strings within our text variable to ensure that the data is relevant and query has captured the phenomena we are aiming to model.

4.3.2 Data Cleaning/Processing:

Again, as we may not always be responsible for data collection, we can expect that our data may contain unhelpful or even problematic information which is often the result of data being unwillingly bought in by the query. Our job at this stage is to minimize the amount of unhelpful data existing in our corpus to ensure our findings are accurate as well as appropriate for the data which we will be modelling.

Optimal procedures for spam detection and removal are covered in more detail [here]*will include link when data cleaning section is complete.* However, there are steps the user absolutely must take to ensure that the text variable which will be provided to the sentence transformer model is clean and concise so that an accurate embedding process can take place upon our documents. This includes the removal of:

- Hashtags #
- User/Account Mentions
- URLs or Links
- Emojis
- Non-English Characters

Often, we might also choose to remove punctuation and/or digits, however in our provided example, we have not done so. There are also things to beware of such as documents beginning with numbers that can influence the later processes, so unless we deem them necessary we should remove these where possible to ensure no inappropriate grouping of documents takes place based on these minor similarities. This is because when topic modelling, we aim to capture the pure essence of clusters which is ultimately defined by the underlying semantic meaning of documents, as apposed to any similarities across the chosen format of said documents.

4.3.3 Sentence Transforming/Embedding:

Once we are happy with the cleanliness and relevance of our data, including the processing steps we have taken with our chosen text variable, we can begin embedding our documents so that we have a numerical representation that can later be reduced and visualised for each. Typically, and in this case we have used already pre-trained sentence transformer models that are hosted on Hugging Face, such as `all-mpnet-base-v2` which is the specific model we had decided to use in our AI Conversation Landscape example. This is because during that time, the model had boasted great performance scores for how lightweight it was, however with models such as these being open-source, community-lead contributions are made to further train and improve model performance which means that these performance metrics are always increasing, so one may wish to consult the Hugging Face leaderboard, or simply do some desk research before settling on an ideal model appropriate for their own specific use case.

While the previous steps taken might have involved using R and Rstudio and making use of SHARE's suite of data cleaning, processing and parsing functionality, the embedding process will need to be completed using Google Colab. This is to take advantage of their premium GPUs and high RAM option, as embedding documents can require large amounts of compute, so much so that most fairly competent machines with standard tech specs will struggle. It is also worth noting that an embedding output may depend on the specific GPU being utilized as well as the version of Python that Colab is currently running, it's good practice to make note of both of these specifics, along with other modules and library versions that one may wish to use in the same session, such as `umap-learn` (you may thank yourself at a later stage for doing so). To get going with sentence transformers and for downloading/importing a model such as all-mpnet-bas-v2, there are step-by-step guides purposed to enable users with the know-how to use them and deal with model outputs upon the Hugging Face website.

4.3.4 Dimension Reduction:

At this stage, we would expect to have our data cleaned along with the representative embeddings for each document, which is output by the sentence transforming process. This next step, explains how we take this high-dimensional embeddings object and then simplify/reduce columns down enough to a more manageable size in order to map our documents onto a semantic space. Documents can then be easily represented as a node and are positioned within this abstract space based upon their nature, meaning that those more semantically similar will be situated closer together upon our two (or sometimes three-dimensional) plot, which then forms our landscape.

There are a number of ways the user can process an embeddings output. Each method has its own merits as well as appropriate use cases, which mostly depend whether the user intends to focus on either the local or global structure of their data. For more on the alternative dimension reduction techniques, the BERTopic documentation provides some further detail while staying relevant to the subject matter of Topic Modelling and NLP.

Once we have reduced our embeddings, and for the sake of staying consistent to the context of our given example, lets say we have decided to use Uniform Manifold Approximation and Projection (UMAP), a technique which is helpful for when we wish to represent both the local and global structures of our data. The output of this step should have resulted in taking our high dimensional embedding data (often 768 columns or sometimes more) and reduced these values down to just 2 columns so that we can plot them onto our semantic space (our conversation landscape plot), using these 2 reduced values as if to serve as X and Y coordinates to appropriately map each data point, we often name these two columns `V1` and `V2`.

At this stage, we can use the `LandscapeR` package to render a static visualisation of the entire landscape, and we can select the desired colour of our nodes by making use of the `fill_colour` parameter. In this instance, we've mapped our documents onto the semantic space represented as nodes using columns `V1` and `V2` and coloured them a dark grey.

```
data %>%
  LandscapeR::ls_plot_static(x_var = V1,
                             y_var = V2,
                             fill_colour = "#808080")
```

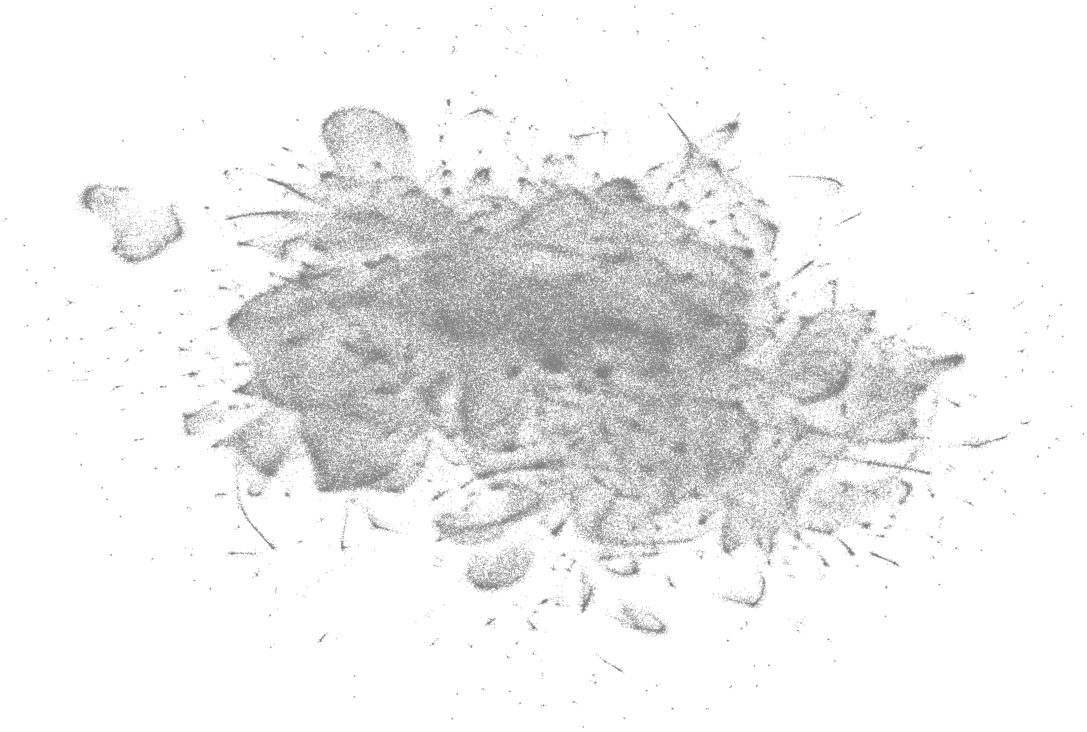


Figure 4.2: Grey Colourless Landscape Plot from an AI Landscape Microsoft Project - Q2 FY24

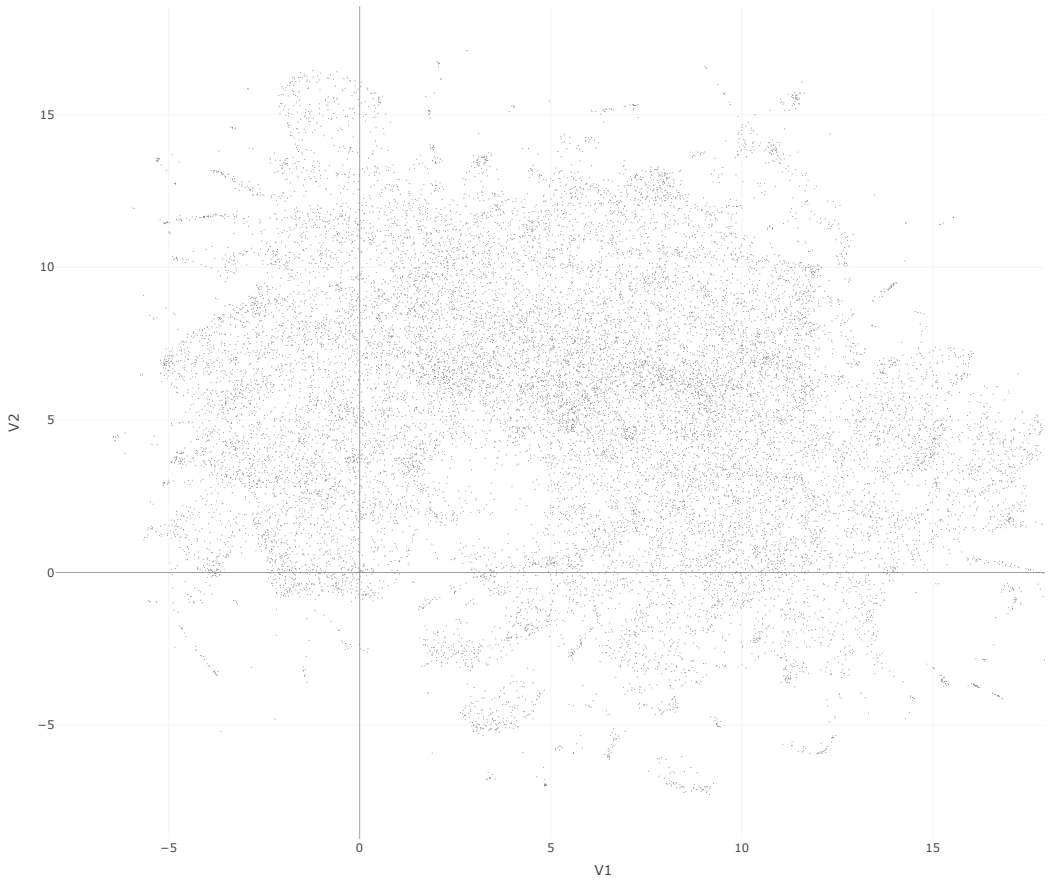
It's worth pointing out, that there are a number of ways for the user to interactively explore the landscape at this stage by scanning over each node, checking the documents contents. This helps the user to familiarise with each region of the landscape before clustering. The `plotly` package serves as a user friendly means for this purpose, helping us gather a 'lay of the land' and identify the dense and not so dense sections of our data.

> **Note**
>
> *This shows just a 20K sample from our data, which is done only to comply with data size limits on GitHub and to be more conservative with compute and memory usage. Here, we also use a message column with breaks every 10 words to ensure the visual is neater.*

```
data %>% plotly::plot_ly(
  x = ~V1,
  y = ~V2,
  type = 'scatter',
```

```
  mode = 'markers',
  marker = list( color = '#808080', size = 1),
  text = ~paste('Message: ', message_with_breaks),
  hoverinfo = 'text'
)
```

```
PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installe
```


15
10
5
0
−5
V2
−5
0
5
10
15
V1

4.3.5 Topic Modelling/Clustering:

The final steps taken are arguably the most important, this is where we will define our documents and simplify our findings byway of scientific means, in this case using Topic Modelling.

There are a number of algorithms that serve this purpose, but the more commonly used clustering techniques are KMeans and HDBSCAN. However, the example we have shown uses KMeans, where we define the number of clusters that we would expect to find beforehand and perform clustering on either the original embeddings object output by the sentence transformer model, or we can reduce those embeddings to something much smaller like 10 dimensions and cluster documents based on those. If we were to opt for HDBSCAN however, we would allow the model to determine how many clusters were formed based on some input parameters such as `min_cluster_size` which are provided by the user. For more on these two techniques and when/how to use them in a topic modelling setting, we can consult the BERTopic documentation once more.

It's also worth noting that this step requires a significant amount of human interpretation, so the user can definitely expect to partake in an iterative process of trial and error, trying out different values for the clustering parameters which determine the models output, with hopes of finding the model of best fit, which they feel accurately represents the given data.

In practise, this visualisation can be derived using our original data object with topic/cluster information appended, as well as the original `V1` and `V2` coordinates that we had used previously. To ensure our topics are coloured appropriately, we can create and use a named character vector and some additional `ggplot2` syntax to manually assign topics with specific hex codes or colours.

```
# assign colours to topics
topic_colours <- c("Ethics & Regulation" = "#C1E1C1",
                   "Technological Innovations" = "#6e88db",
                   "AI in an Artistic Domain" = "#7e2606",
                   "Cultural & Social Impact" = "#ff6361",
                   "Business & Wider-Markets" = "#063852",
                   "Future of Learning & Personal Growth" = "#ffa600",
                   "Future of Work & Security" = "#9e1ad6"
                   )
```

```
data %>%
  LandscapeR::ls_plot_group_static(x_var = V1,
                                   y_var = V2,
                                   group_var = topic_name) +
  ggplot2::scale_colour_manual(values = topic_colours) # colour nodes manually
```

Figure 4.3: Segmented Colourful Landscape Plot from an AI Landscape Microsoft Project - Q2 FY24

4.4 Downstream Flourishes

With the basics of each step covered, we will now touch on a few potentially beneficial concepts worth grasping that may help us overcome anything else that may occur when working within the Conversation Landscape project domain.

4.4.1 Model Saving & Reusability:

Occasionally, a client may want us to track topics over time or perform a landscape change analysis. In these cases, we need to save both our Dimension Reduction and Clustering models so that new data can be processed using these models, tp produce consistent and comparable results.

This requires careful planning. When we initially reduce embeddings and perform clustering, we use the `.fit()` method from `sklearn` when either reducing the dimensions of or clustering on the original embeddings. This ensures that the models are trained on the data they are intended to represent, making future outputs comparable.

We had earlier, mentioned, that it is crucial to document the versions of the modules and Python interpreter used. When we reduce or cluster new data using our pre-fitted models, it is essential to do so with the exact same versions of important libraries and Python. The reason is that the internal representations and binary structures of the models can differ between versions. If we attempt to load and apply previously saved models with different versions, we risk encountering incompatibility errors. By maintaining version control and documenting the environment in which the models were created, we can ensure the reusability of our models. Overall, this practice allows for us to be accurate when tracking and comparing topics and noting any landscape changes.

4.4.2 Efficient Parameter Tuning:

When we're performing certain steps within this workflow, more specifically the Dimension Reduction with likes of UMAP, or if we were to decide we'd want to cluster using HDBSCAN for example, being mindful of and efficient with tuning the different parameters at each step will definitely improve the outcome of our overall model. Therefore, understanding these key parameters and how they can interact will significantly enhance the performance of the techniques being used here.

4.4.2.1 Dimension Reduction with UMAP:

n_neighbors: This parameter controls the local neighborhood size used in UMAP. A smaller value focuses more on capturing the local structure, while a larger value considers more global aspects. Efficiently tuning this parameter involves considering the nature of your data and the scale at which you want to observe patterns.

min_dist: The min distance argument determines quite literally how tight our nodes are allowed to be positioned together within our semantic space, a lower value for this will mean nodes will be tightly packed together, whereas a higher number will ensure larger spacing of data points.

n_components: Here is where we decide how many dimensions we wish to reduce our high-dimensional embeddings object down to, for visualisation we will likely set this parameter to a value of 2.

4.4.2.2 KMeans CLustering

n_clusters: KMeans is a relatively simple algorithm compared to other methods and components, requiring very little input. Here we just provide a value for the number of clusters we wish to form, this will either be clusters in the embeddings or a smaller, more manageable reduced embeddings object as mentiioned previously.

4.4.2.3 HDBSCAN Clustering:

min_samples: This parameter defines the minimum number of points required to form a dense region. It helps determine the density threshold for clusters and can determine how conservative we want the clustering model to be. Put simply, a higher value can lead to fewer, larger clusters, while a lower value can result in more, smaller clusters.

min_cluster_size: This parameter sets the minimum size of clusters. Like `min_samples` it can directly influence the granularity of the clustering results. In this case, smaller values allow the formation of smaller clusters, while larger values prevent the algorithm from identifying any small clusters(or those below the size of the provided value). It's worth noting that the relationship between `min_samples` and `min_cluster_size` is crucial. `min_samples` should generally be less than or equal to `min_cluster_size`. Adjusting these parameters in tandem helps us to control the sensitivity of HDBSCAN, and for us to define what qualifies as a cluster.

4.4.2.4 Tip: Try starting with the default value for all of these parameters, and incrementally adjust based on the desired granularity or effect of any that we wish to amend.

4.4.3 Supporting Data Visualisation:

Once we have our landscape output, as shown in Final output of project, we will inevitably need to display some further information regarding our topics, most commonly; Volume over Time (VOT) and Sentiment Distribution for each.

When doing so, we would ideally keep some formatting consistencies when plotting, as we mentioned previously, the colouring of our topics must remain the same throughout so that they match up with any previous representations in existing visuals such as the landscape output. We would also want to ensure that any plot we create orders our topics by volume or

at least in the same order throughout our project. We can order our topics in terms of volume easily with just a few lines of code.

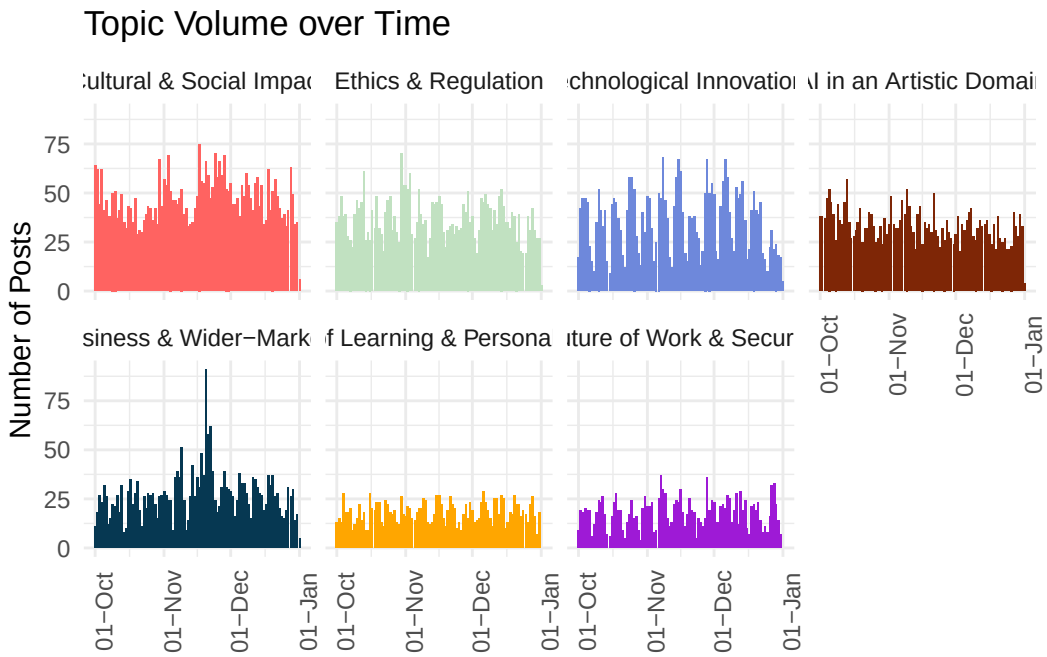
First, we'll make sure to set the factor levels of our topics by using `dplyr::count()` on the `topic_name` column, and setting the levels feature of the `factor()` base function based on the counted output.

```
# sort topics by order of volume
topic_order <- data %>% dplyr::count(topic_name, sort = TRUE)
# set levels determined by volume of topic, this orders the group variable for plotting
data <- data %>%
  dplyr::mutate(topic_name = factor(topic_name, levels = topic_order$topic_name))
```

4.4.3.1 Topic Volume over Time

Starting with volume over time, we often choose to render a faceted plot that includes all topics and their VOT for comparison. We can do so by using functionality found in packages such as `JPackage` for this.

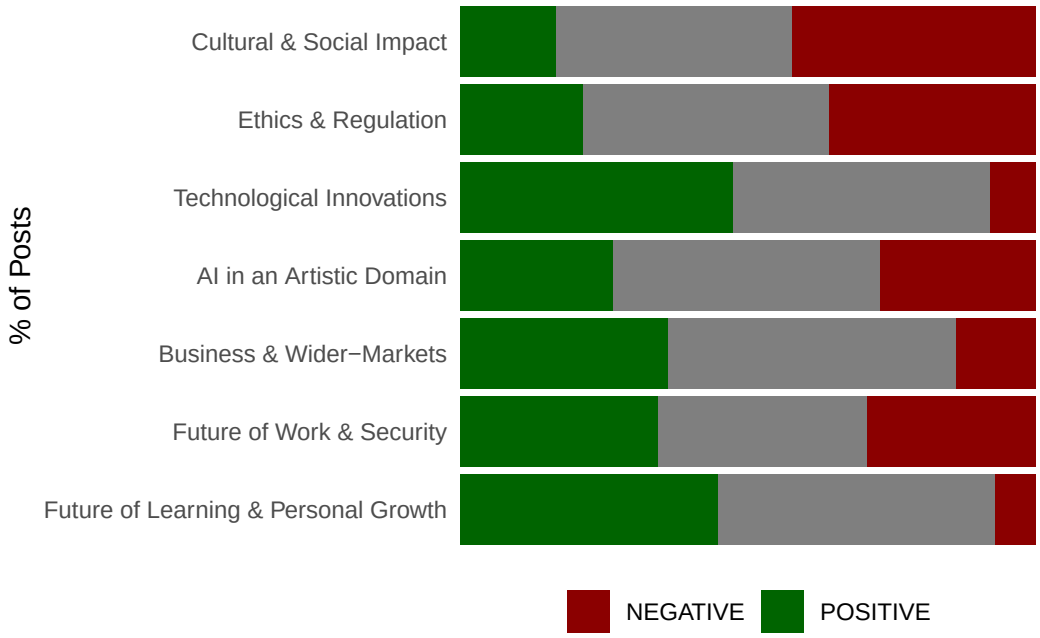
```
# plot topic volume over time using 'plot_group_vol_time()' function
data %>%
  JPackage::plot_group_vol_time(group_var = topic_name,
                                date_var = date,
                                unit = "day",
                                nrow = 2) +
  ggplot2::scale_fill_manual(values = topic_colours) # apply colours manually!
```



4.4.3.2 Topic Sentiment Distribution

Next, we might want/need to break each of our topics out by their sentiment distribution to help shine light on any of particular interest or to help us tell a more refined story using our topic model. This can be done by using the `dr_plot_sent_group()` function of the `DisplayR` package.

```
data %>%
  DisplayR::dr_plot_sent_group(group_var = topic_name,
                               sentiment_var = sentiment,
                               "percent", bar_labels = "none",
                               sentiment_colours = c("POSITIVE" = "darkgreen",
                                                     "NEGATIVE" = "darkred"))
```



4.4.3.3 Alternative Visualisations

While the two visuals we have displayed so far are relatively basic and commonly used, this does not mean that we won't require alternative methods to display topic-level information. Often, we may render n-grams per topic to display the relationships that exist between terms/phrases, and we may create plots to showcase things such as data source or social network/platform distributions across topics.

Finally, it's worth noting that the need for specific data visualisation methods entirely depends on the project domain and brief, as well as any outcomes/findings derived throughout. This means we ought to be flexible in our approach to utilising any technique that may assist with strengthening our understanding of the data and/or supporting our analyses.

Part III

Project work

5 A Data Science project

5.1 What is a Data Science project?

Aside from the obvious definition of a project (a piece of work planned and executed to achieve a particular aim- in this case facilitate a client's needs), what this section is referring to is the structure and usage of a coding project.

5.2 Where are projects saved/located?

All projects need to be saved onto the Google Drive. We have our own Data Science section, where we save our project and internal work (code, data, visualisations etc), which is in the filepath:

`Share_Clients/data_science_project_work/`

You should get access to this directory straight away.

Within the `data_science_project_work` directory there are subdirectories of all of our clients, such as `data_science_project_work/microsoft`, `data_science_project_work/dyson` etc.

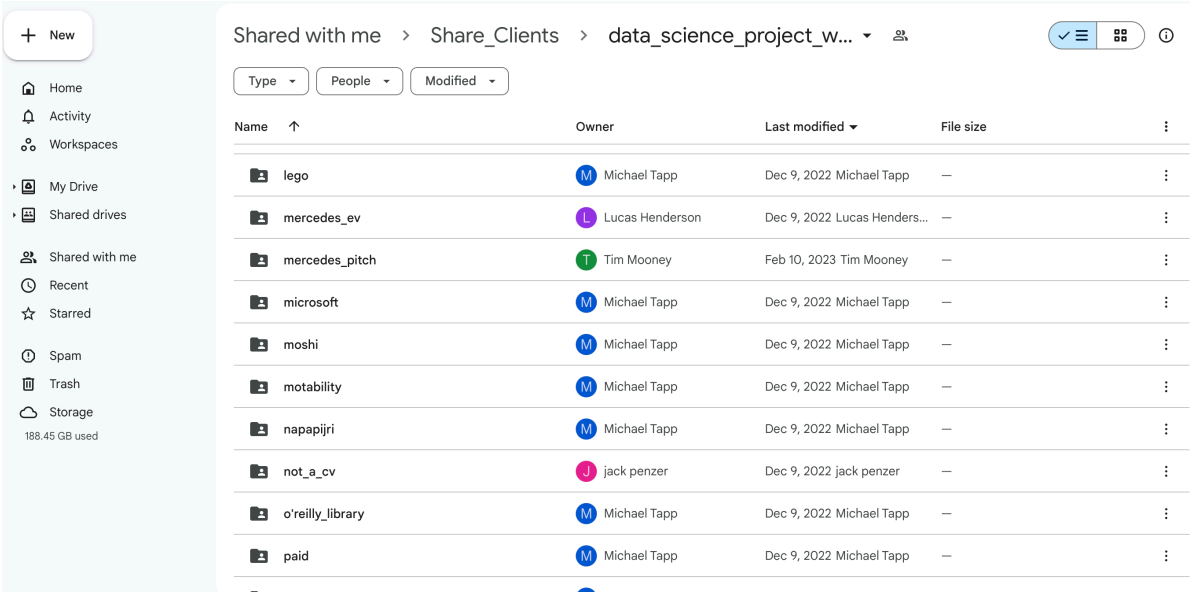


Figure 5.1: Screenshot of the `data_science_project_work/` directory with client-specific sub-directories

> **File paths**
>
> You will see that we refer to the location of directories mainly by their filepath, with the above screenshot of the Google Drive just for full transparency and clarity.
> There are no two ways about it, getting familiar with working with filepaths in the command line (or in a script) is non-negotiable, but will become second nature and you will be tab-completing filepaths in no time at all!

5.3 RStudio Projects

We are primarily an `R` focused team, and as such we utilise RStudio projects to help keep all the files associated with a given project together in one directory.

To create a RStudio Project, click File > New Project and then follow the below steps, but call the directory the name of the project (if a Microsoft project, appended by the project number) rather than 'r4ds'. Be sure to make sure the option 'Create project as subdirectory of' is the client directory on the Drive (in the case of Microsoft, this is `Share_Clients/data_science_project_work/microsoft/project_work/`).

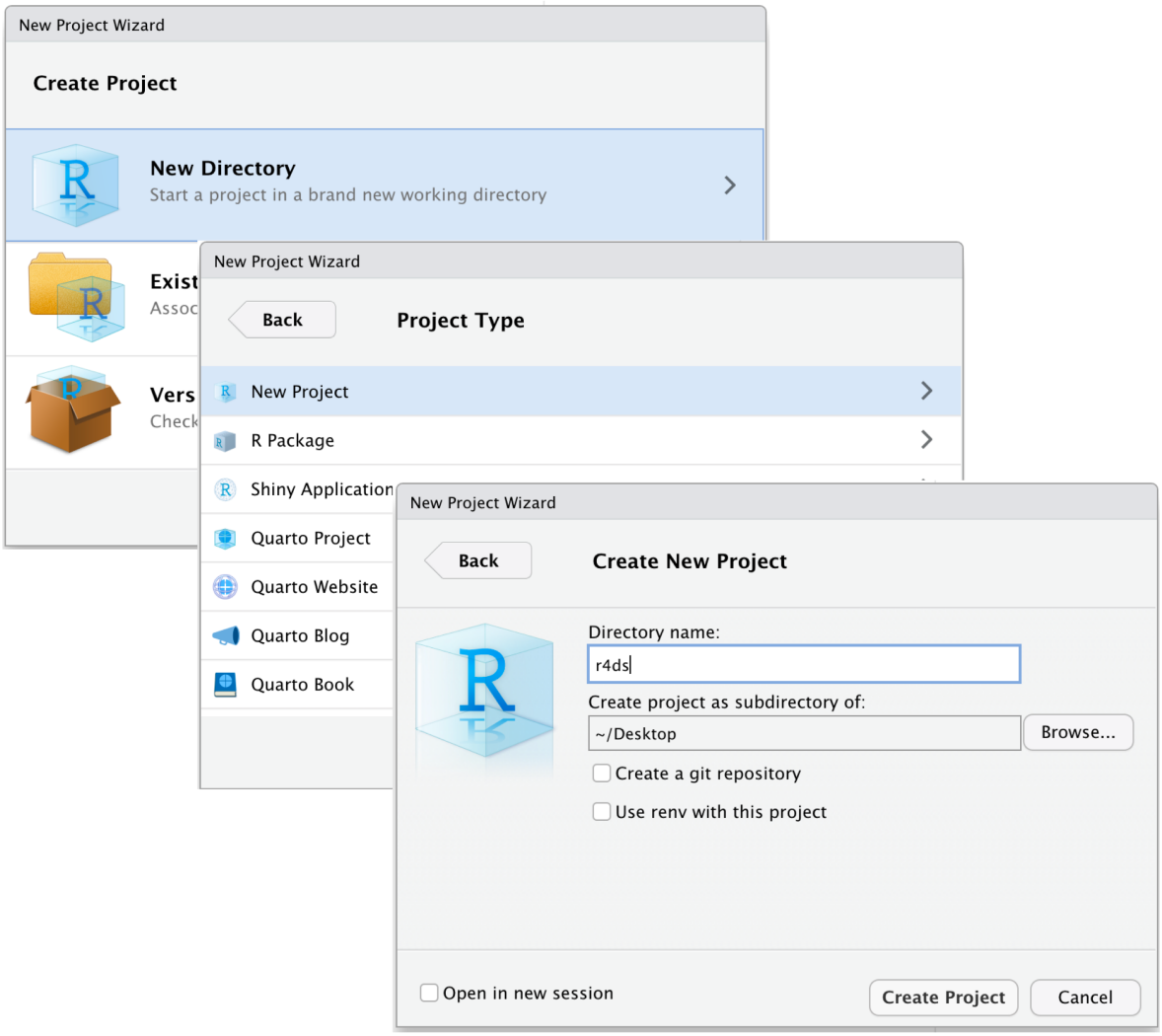


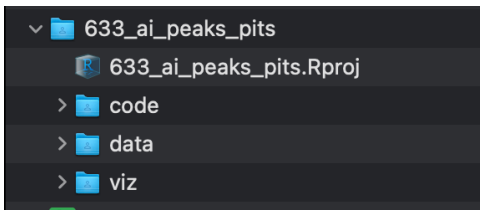
Figure 5.2: Steps to create a new project, taken from R for Data Science (2e) https://r4ds.hadley.nz/workflow-scripts.html#projects

Once this process is complete, there should be a new project folder in the client directory, with a `.Rproj` file within it.

If this is your first time using RStudio Projects, we recommend reading this section within the R for Data Science book, to familiarise yourself with some more intricacies of Project work within R (such as relative and absolute paths) which we would not do justice summarising here.

5.4 Components of a DS project

DS projects consist of a parent project directory, with an associated `.Rproj` file, and three compulsory subdirectories `code`, `data`, and `viz` (all of which are made manually).



Whilst there are no prizes for what goes in each subdirectory, it can be useful to have a structure in place to facilitate workflow ease.

5.4.0.1 `code`

Within the `code` subdirectory is where all scripts should be kept. We utilise `.Rmd` (R Markdown) documents rather than basic `.R` scripts for our code.

We do this for a few reasons, but the main benefits include:

- It acts as an environment in which to do data science- we can capture not only what we did, but importantly *why* we did it
- We can easily build and export a variety of different output formats from a R Markdown document (PDF, HTML, slideshow etc)

As part of our commitment to literate programming, there are some good practices that we can implement at this level of abstraction.

Firstly, do not have extremely long `.Rmd` documents, as this is no good for anybody. Instead split up your documents into different sections based on the purpose of the code.

Whilst this can be a bit subjective, a good rule of thumb is to have a separate `.Rmd` for each aspect of a workflow. For example, we might have one `.Rmd` for reading in raw data, another for cleaning the data, another for EDA, and another for performing topic modelling etc.

We should also follow the tidyverse style guide in the naming of files, which states:

> If files should be run in a particular order, prefix them with numbers

Therefore it makes sense to prefix our files, as we must load in the raw data before we can clean the data, and we must clean the data before we can perform certain analyses etc.

So we might have something like `00_load_data.Rmd`, `01_clean_data.Rmd`, `02_topic_modelling.Rmd`.

5.4.0.2 `data`

`data` is where we save any data file that comes from a project.

The vast majority of projects will involve analysing an export from a social listening platform, such as Sprinklr. Analysts will save the export in the form of `.csv` or `.xlsx` files on the Drive (not within the Data Science section). As Sprinklr limits its exports to 10k rows of data per export file, we often are presented with 10s/100s of files with raw mentions. Therefore once we read these files into R, it is a good opportunity to save them as an `.Rds` in the `code` folder using the function `write_rds()` to avoid having to reread the raw excel or csv files in again.

It is within `data` where you would also save cleaned datasets and the outputs of different analyses (not visualisations though). This is not limited to `.Rds` files, but could also be word documents, excel spreadsheets etc.

As projects get more complex with many analyses, it can be easy to clutter this subdirectory. As such, it is recommended to make folders within `data` to help maintain structure. This means it is easy to navigate where cleaned data is because it will be in a folder such as `data/cleaned_data` and a dataframe with topic labels would be in `data/topic_modelling`.

> Save liberally
>
> Generally speaking, space is cheaper than time. If in doubt, save an intermediate dataframe after an analysis if you *think* you'll need it in the future. It is better to run an analysis once and save the output to never look at it again, than to run an analysis, not save the output, and then need to rerun the analysis the following week.

5.4.0.3 `viz`

Any visualisation that is made throughout the project should be saved here. Again, this directory should be split into separate folders to keep different analyses separate, navigable, and clear. This is especially useful if there is are visualisations being made of the same analysis mapped over different variables or parameters, or if the project involves the analysis of separate products or brands.

For example, the below shows a screenshot of a `viz` folder for a project that looked at three products. Within `viz` the plots for each brand are in their own folder, and within each brand (`chatgpt`, `copilot`, `gemini`) there are further folders to split up the type of visualisations created (`area_charts`, `eda` etc), with even a third level of subdirectory (`area_charts/peaks` and `area_charts/pits`).

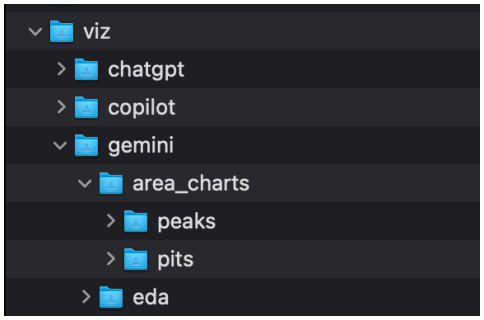


Figure 5.3: Example viz directory hierarchy for a Peaks and Pits project

6 Project key players

6.1 Insights Analyst

Analysts add the bit of human-insight sparkle to our projects. They work closely with the client and stakeholder to help frame our findings so they are suitable for the clients business needs. At a high level, we may say "Our clustering analysis identified five distinct regions of conversation based on the semantics of the language used" whereas an analyst would translate that to "We have five key conversational themes that can be targeted with tailored marketing strategies to boost product reach on socials". Though this does vary on a project by project basis and we often have to act as a conduit between the science and the client too.

Insight analysts are who work closely with Sprinklr and other social listening platforms to obtain the data we analyse. They will craft queries to pull the relevant data from a variety of sources and save the data on the Drive for us to access and do science on.

6.2 Account Manager

Account Managers (AMs) are the point of contact between our business and the client, bridging the gap between the technical teams (in our cases DS or Insights) and the client/stakeholder. They will arrange meetings with the client, help us understand the client's needs and business objectives, and coordinate project logistics, timelines, and deliverables. As project deadlines approach, account managers will help QA our deliveries (normally in the form of a PowerPoint or Keynote presentation), providing valuable opinion from a non-technical background (it can be easy for us to get stuck in the weeds and forget that stakeholders do not know as much about data as we do). Broadly, AMs make sure both us as a company, and the client, are held accountable for the work we are contracted to do.

6.3 Data/Insight Director

Depending on the project, there will be a Data Director or Insight Director involved on the project too. You will notice that they will normally be resources on Float Whilst not working on the nitty gritty of the project, they are there to help steer the project in the appropriate

direction based on the clients business needs. They will also be checking the final delivery as it is created, making sure the deliverables and story we have thread is suitable and valid.

Part IV

Development

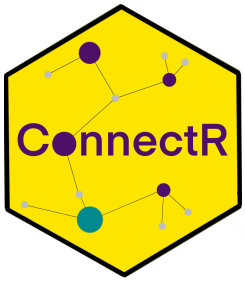
7 Our Packages

We have a suite of R packages that have been developed internally. They all serve different purposes on a project, but together aim to empower the SAMY Alliance. We don't license the software to clients. What we sell is the knowledge that they can produce.

7.1 ParseR

ParseR is the collective name for the techniques SAMY uses for text analysis. It's primarily based on the tidytext philosophy and the analysis is normally carried out in R.

7.2 ConnectR



ConnectR is our package for network analysis. It helps the user find important individuals by graphing retweets and important communities by graphing mentions.

7.3 SegmentR

SegmentR is the collective name for the techniques SAMY uses to find latent groups in data.

7.4 BertopicR

BertopicR is our package which allows access to BERTopic's modelling suite in `R` via `reticulate`.

7.5 LandscapeR

LandscapeR is our package for exploring text data which has been transformed into a navigable landscape. The package makes use of cutting-edge language models and their dense word embeddings, dimensionality reduction techniques, clustering and/or topic modelling as well as Shiny for an interactive data-exploration & cleaning UI.

If the conversation has been mapped appropriately, you will find that mentions close together in the Shiny application/UMAP plot have similar meanings, posts far apart have less similar meanings. This makes it possible to understand and explore thousands, hundreds of thousands, or even millions of posts at a level which was previously impossible.

7.6 LimpiaR

LimpiaR is an R library of functions for cleaning & pre-processing text data. The name comes from 'limpiar' the Spanish verb'to clean'. Generally when calling a LimpiaR function, you can think of it as 'clean…'.

LimpiaR is primarily used for cleaning unstructured text data, such as that which comes from social media or reviews. In its initial release, it is focused around the Spanish language, however, some of its functions are language-ambivalent.

7.7 DisplayR

DisplayR is our package for data visualization, offering a wide array of functions tailored to meet various data visualization needs. This versatile package aims to improve data presentation and communication by providing visually engaging and informative graphics.

7.8 HelpR

HelpR is SAMY's R package for miscellaneous functions that can come in handy across a variety of workflows.

As you progress through your data science journey, you may take an interest in developing your own package. Depending on your previous experience developing software, this might be

daunting, but don't worry none of us had much experience building packages when we joined; we all learned on the job - and so can you. If you want to.

See the Package Development document for more information.

8 Package Development

One of the beautiful things about Data Science at SAMY is that you get to choose your path between research, development, or a hybrid. If you're reading this, you've probably decided that you want to explore some development - what better way to start than building your own package?

> **Note**
>
> For a high-level overview of our existing packages, refer to the our packages document.

We build packages because they are convenient ways to share our code & data and democratise access to our tools & workflows - besides, that Google Doc of functions is getting heavy, and copying & pasting code from project to project is getting tiresome. Eventually you'll want the familiar `library()` syntax.

Historically our packages have been built in R for two key reasons:

1. The profile/experience of people in the team
2. The ecosystem for building packages is well-maintained and documented

In recent times we have moved to more of a hybrid approach between R & Python - the former we find to be considerably more easy to use for data wrangling and visualisation, and the latter for modelling and anything to do with LLMs. Internal development for Python has lagged behind R, but we expect this to change over time as we seek to be tool agnostic and focus on the right tool for the job at hand.

> **Reticulate**
>
> Reticulate is an R package that allows us to import Python packages and functions in R. Currently this is a one-way street - we can't use reticulate to import R functions and packages. This has impacted our decision in the past, e.g. with BertopicR. We envisioned Insight Analysts using BertopicR as a drop-in or replacement for topic modelling with SegmentR. Weighing up the additional difficulty in development vs the time and resource necessary for Analysts to learn Python as well as R, we opted for reticulate.
> Using reticulate requires managing Python environments from R, this leads to difficulties of its own.

9 R

Here we'll look at how to get off the ground in R using the R package stack - {usethis}, {pkgdown}, {devtools}, {testthat} and {roxygen2}.

9.1 Building your own package

The (nearly) DIY way:

This check-list should get you **most of the way** there, but it's always possible that we've forgotten something or there has been a disturbance in the force a change in the package ecosystem. When this happens, open up an issue or submit a PR and help the next person who comes along.

- ☐ Create a new repository on GitHub
- ☐ Clone the repository

> Folder management
>
> Create a folder at your home directory named 'git_repos' and store all of your repositories here:

- ☐ Open RStudio and call `usethis::create_package()`
- ☐ `usethis::use_git()` in the console and commit files
- ☐ Check git has been set up with `usethis::git_sitrep()` (or `git status` in the terminal)
- ☐ Set an upstream branch e.g. `git remote add upstream <link_to_repo_main>` in the terminal
- ☐ `usethis::use_vignette()` to add a vignettes folder and start your first vignette
- ☐ Add individual functions/scripts with `use_r("script_name")` - these appear in your R/ folder
- ☐ Document each function following roxygen2's structure Roxygen2 guidelines
- ☐ Call `use_package()` whenever you use an external package inside your package.

DESCRIPTION

Your package will now have a DESCRIPTION file, add external packages your package requires to Imports. Add additional packages used in vignettes to Suggests. - But be careful, it's generally not advisable to use packages just for vignettes!
You can use the `usethis::use_latest_dependencies()` to add recent versions to your packages, but beware this can be restrictive. Ideally you would add the minimum package version necessary to run your code.

- ☐ `usethis::use_*_license()` - default to usethis::use_mit_license()
- ☐ `usethis::use_testthat()` and `use_test("script_name")` to start writing units tests for your functions and add testthat to suggests.
- ☐ Call `usethis::use_readme_rmd()` to allow for markdown code chunks in your readme - just remember to `devtools::build_readme()` when you're done.
- ☐ Call `usethis::use_news_md()`
- ☐ When you're ready to add a website, call `usethis::use_pkgdown()` `pkgdown::init_site()`, `pkgdown::build_home_index()`, `pkgdown::build_search()`, `pkgdown::build_reference()`, `pgkdown::build_articles()`, and then `pkgdown::build_site()`
- ☐ Add each function to pkgdown.yml's reference section (we recommend viewing a working yml file from one of the other packages to get you started).

The Easy Way (Tidy functions) Read through the usethis-setup guide and then use the `usethis::create_tidy_package()` to create a package with some guardrails.

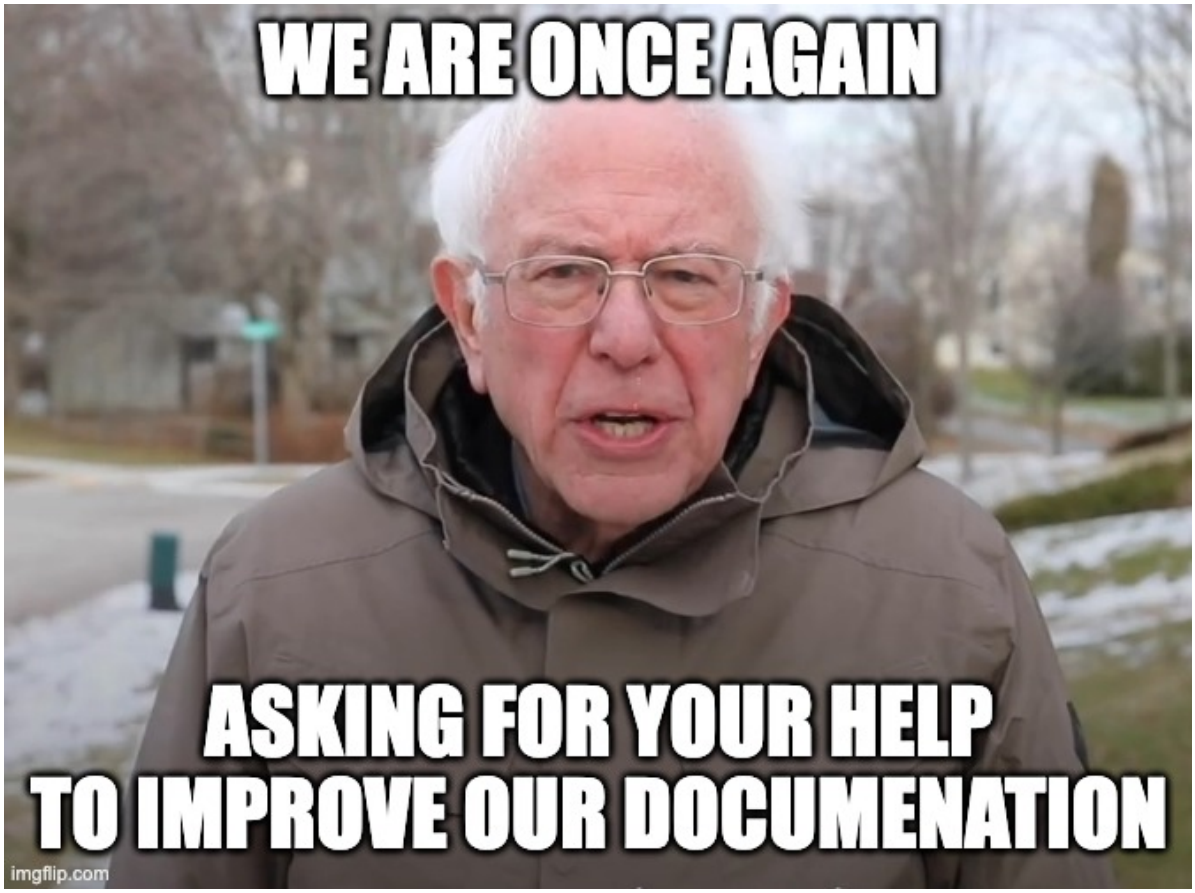
Guardrails or no guardrails?

The `usethis::create_tidy_package()` function is a helpful abstraction, but it will be better for your long-term development if you know how to do this stuff without the abstraction. That way, when you need to fix something, or do something slightly different than the prescribed way, you'll have a better chance of success.

Want to go deeper? Check out the R Packages Book, we recommend skimming first and then using it as a reference manual.

9.2 Development workflow

Once you've built the package there are some things you will want to do regularly to ensure your package stays in good shape. This is by no means an exhaustive list - be sure to add your tips & tricks as you amass them.



- ☐ Run `testthat::test_package()` often to check for regressions in your code
- ☐ Run `devtools::check()` occasionally to make sure you haven't made any obvious mistakes - try to keep notes, warnings and errors to 0!
- ☐ Use `devtools::load_all()` to reload the package when you've made changes. (`devtools::document()` also calls `load_all()` when called)
- ☐ Run `roxygen2::roxygenise(clean = TRUE)` if your documentation doesn't look as you expect after
- ☐ Use `pkgdown::build_site()` when you expect to see changes in your package's website
- ☐ Use `pkgdown::clean_site()` and `pkgdown::build_site()` when expected changes aren't reflecting in your preview

9.3 Contributing to existing packages

- ☐ Pull current state of repo/package from origin

- ☐ Create a new branch, can use `usethis::pr_init()` from usethis to make this a bit easier, otherwise `git checkout -b "branch_name"`
- ☐ Run `devtools::test()` `devtools::check()` at regular intervals, keep errors, warnings, notes down to minimum
- ☐ Build out logic for new changes, add to R/ where necessary. `usethis::use_r()` function to add new scripts properly
- ☐ Build out tests for new logic in tests/
- ☐ Ensure function-level documentation is added to any new logic, including @title, @description, @details, @param, @returns, @examples and @export if function is to be exported, or @internal otherwise. Let roxygen2 take care of @usage.
- ☐ Keep re-running tests and check!
- ☐ If you're introducing something new, update package-level documentation e.g. vignettes and/or readme explaining what you've introduced and how it should be used. Provide examples where possible. If you're building out a new capability you may need a whole new vignette, use the `usethis::use_vignette()` function.

> What are vignettes?
>
> Vignettes are long-form guides that provide in-depth documentation for your package. They go beyond the basic function documentation and explain how to use the package to solve specific problems, often with detailed examples and code. Vignettes showcase your package's full range of capabilities and help users understand how to effectively utilise its features

- ☐ If you're updating legacy code, check that vignettes are up-to-date with the changes you've made - we want to avoid code-documentation drift where possible.
- ☐ Add your function to the reference section in _pkgdown,yml if it's being exported.
- ☐ Add data objects, `.Rhistory`, `*.Rproj`, `.Rprofile`, `.DS_Store`, to `.gitignore`
- ☐ Run `pkgdown::clean_site()` and `pkgdown::build site()`, visually inspect each section of the site

Pull request when ready.

9.3.1 Code

Generally code should sit in the R/ folder, you can choose between a script per function or use scripts as modules, where a module is a particular use case, or logical unit. Historically we sided on the former, but as a package grows it can become difficult to manage/navigate, and there can be a decoupling of logic. Ultimately this is a matter of taste in R.

9.3.1.1 Exercises - code

> ⚠ Exercises
>
> You may need to consult external resources to answer the exercises, we've tried to provide links to help you along the way, but we encourage you to embrace the joy of discovery and find relevant sources/fill in the gaps where necessary!

1. What are the practical differences between `.gitignore` and `.Rbuildignore`?

- What objects should go in `.gitignore` but not `.Rbuildignore`, and vice versa?

2. What does the DESCRIPTION file do?
3. Write your own description for each of the following packages, detailing what they are for where they sit in the R package stack:

☐ testthat
☐ roxygen2
☐ devtools
☐ pkgdown
☐ usethis

9.3.2 Tests

> In a perfect world, every dog implementation detail would have a home test and every home test would have an implementation detail.

There is a balance to be struck between testing *absolutely everything* and testing what needs to be tested. Before we get into the finer details, let's establish why we're writing tests in the first place. The first reason for writing tests is to help you write software that works. The second reason is to help you do this fast, and with confidence.

Testing is not to prove that your code has no bugs, or cannot have any bugs in the future. Whenever you do find a bug, or someone reports one, write a test as you fix the issue.

For more information and another opinion, check out the R Packages testing section, and the Testing document

> Tip
>
> Don't let testing paralyse your development process, they're there to help not hinder. As a rule-of-thumb, if your tests for a function are more complex than your function, you've gone too far.

9.3.3 Documentation

We use {roxygen2} tags to document our functions. Visit the documenting functions article for a primer.

Using the roxygen2 skeleton promotes consistent documentation, check out a function's help page (e.g. `?ParseR::count_ngram`) to see how rendered documentation looks - do this regularly with your own functions.

We tend to find our documentation could always be better, more complete. You can't hope to cover **everything** a user could do with your function, but make sure it's clear from the documentation what your function is for and what its primary uses are.

```
#' @title Visualise the top terms by frequency for a grouping variable
#'
#' @description A network visualisation, where the big nodes are levels of
group_var e, e.g. topic -> topic_1, topic_2 etc. connected by edges to the
small nodes which are the most frequently seen terms for each level of the
grouping variable
#'
#' @details The main idea of this function is to help identify which groups
have similar terms associated with them - big nodes will placed close by to
the other big nodes they share terms with, if a big node shares no other terms
with another big node they will be placed far apart.
#'
#' It's important to communicate how many of the top terms have been selected
for, as if the term "happy" is #18 for group 1, and #21 for group 2, and our
cut off point was 20, we may falsely assume that "happy" is not a term shared
by both groups. Looking further down the list (setting n_terms to 30-40) to
strengthen any inferences made is recommended.
#'
#' @param data A data frame or tibble containing both the text_var and
group_var columns
#' @param group_var A group variable or factor. This could be either; brand,
audience, sentiment or similar
#' @param text_var The text variable assigned to each observation containing
the message or post
#' @param n_terms How many of the highest mentioned terms per group should be
included in the visualization
#' @param text_size An integer stating the desired text size
#' @param with_ties Whether to allow for > n_terms if terms have equal
frequency in `group_var`'s count
#' @param group_colour_map For if the user wants to apply custom colour
mapping to the group variables
#' @param terms_colour What colour should the terms be?
#' @param selected_terms Any terms that the user wishes to colour differently
(should be supplied as a list)
#' @param selected_terms_colour What colour should any selected terms be? This
includes all those defined in the list supplied to the `selected_terms`
argument
#'
#' @return Returns a ggplot network visualization showing the relationship
between terms in a text variable and any group variable in the data, byway of
counting the most frequently used terms in conjunction with each class of the
group variable.
#'
#' @export
#'
#' @examples
#'set.seed(1)
#'viz_group_terms_network(data = ParseR::sprinklr_export,
#'group_var = Sentiment,
#'text_var = Message,
#'n_terms = 20,
#'text_size = 4,
#'with_ties = FALSE,
#'group_colour_map = NULL,
#'terms_colour = "black"
```

> ⚠ Warning
>
> Most people will scroll straight past the @description and @details and go directly to your code examples.

9.3.3.1 Guidelines for commonly-used Roxygen tags

Tag	Description
@title	One-line description of what your function does
@description	A paragraph elaborating on your title
@details	A more detailed description of the function e.g. explaining how its arguments interact, or other key implementation details.
@param	A description of the function's parameters
@return	A description of the function's return value
@examples	Examples of how to use the function
@export	Whether the function is exported or not

9.3.3.2 Exercises - Documentation

1. What is the title of {dplyr}'s `mutate()` function?
2. @examples must be self-contained, create an example that is not self-contained, and one that is.
3. Which package(s) (any programming language) stick out in your mind as being well-documented and easy to use, what did the creators do well?
4. Audit SAMY's R packages, find a function with sub-par documentation and upgrade it. Then fire in a Pull Request!

9.3.4 Data

You're probably going to need some package-level data for your @examples or your vignettes. Before going off and finding or creating a new data set:

1. Check whether you can demonstrate what you need with existing datasets - call `data()` in your console
2. Make sure the dataset you have chosen comes from a package your package explicitly Imports or Suggests

If you still can't find the right dataset, create one!

1. Load the dataset into memory
2. Call `usethis::use_data(dataset_variable_name)`
3. Document the columns

If you choose this route, some interesting problems may lie in wait. Skip to Exercises 1.

To go deeper view the R Packages Dataset Section

Datasets from the {datasets} package come with base R

9.3.4.1 Exercises - Data

1. Why might you add your data artefacts to `.Rbuildignore` or `.gitignore`?
2. Which package does the `diamonds` dataset ship with?

You're probably going to need some data... existing data... adding new usethis::use_data() usethis::use_data_raw()

9.3.5 Website

pkgdown .nojekyll

9.3.5.1 Exercises - Website

1. Explain in your own words what .nojekyll is for.

- Where should it be placed in your package?
- What problems arise when you don't have one?

2.

10 Python

> **Warning**
>
> This document is very much a work in progress, key steps may be missing.

10.1 Folder Setup

The first step in creating a Python package is setting up the project structure. Create a new directory for your project and organize it with the following subdirectories and files:

- ☐ your_package_name/: The main package directory containing your Python modules and code. Make sure to add a `__init__.py` file to the source and any modules.

> **init**.py
>
> Create a directory for your package and place an empty **init**.py file inside it. If your package has sub-packages or modules, create additional directories for them and place **init**.py files in each directory. Import your package or its modules in your Python scripts using the import statement or the from package import module syntax.

- ☐ tests/: Directory for test files to ensure the correctness of your package's functionality.
- ☐ docs/: Directory for storing comprehensive documentation files.
- ☐ setup.py: File specifying package metadata, dependencies, and build instructions.
- ☐ MANIFEST.in: File listing the files to include in the package distribution.
- ☐ requirements.txt: File listing the package dependencies for easy installation.
- ☐ README.md: File providing an overview, installation instructions, and usage examples for your package.
- ☐ LICENSE: File specifying the license under which your package is distributed.
- ☐ .gitignore: File specifying files and directories to ignore in version control.
- ☐ Initialize a Git repository in your project directory and create a new repository on GitHub for collaboration and issue tracking.

10.2 Git - Terminal

To set up a Git repository for your Python package from the terminal, follow these steps:

1. Open your terminal and navigate to the root directory of your Python package using the cd command. For example: `cd /path/to/your/package`
2. Initialize a new Git repository by running the git init command: `git init`
3. Add all the files in your package directory to the Git staging area using the git add command: `git add .`
4. Create an initial commit to save the current state of your package by running the git commit command with a meaningful commit message: `git commit -m "Initial commit"`
5. Add the remote repository URL to your local Git repository using the git remote add command: `git remote add origin <repo_url>`
6. Push your local commits to the remote repository using the git push command: `git push -u origin main`

10.3 Vignettes

- ☐ Write the vignette content in a readable format such as Markdown (.md) or reStructuredText (.rst).
- ☐ Place the vignette file in a dedicated directory within your package, typically named vignettes/ or docs/vignettes/
- ☐ Use a documentation tool like Quarto, Sphinx or MkDocs to convert the vignette file into HTML or PDF format. We advise using Quarto to keep it simple.
- ☐ Configure your package's setup.py file to include the vignette files in the package distribution. Add the vignette directory to the package_data argument of the setup() function.
- ☐ If using Sphinx or mkdocs: Generate the package documentation by running the documentation tool's build command, such as sphinx-build or mkdocs build. This will create the HTML or PDF files for your vignettes.
- ☐ Publish the generated vignette files along with your package distribution. Include them in the source distribution (sdist) and wheel distribution (bdist_wheel) that you upload to PyPI or conda.

10.4 Tests

There are multiple viable frameworks, but for simplicity we recommend Pytest which functions quite similarly to {testthat}.

Pytest has great docs, work through the how-to guides to get up to speed.

11 Continuous Integration/Continuous Deployment

R templates etc. from RStudio Python templates

12 Testing in R

> **Warning**
>
> This document is not rendered by the handbook so some code samples may be out of date/not working. (sorry!)

When developing packages in R, we usually lean on {testthat}. Creating unit tests with {testthat} (leaving aside integration tests for now) is pretty simple, first we write a function which performs some actions, then we write some tests which check that our function still performs those actions - or if following Test Driven Development (TDD) practices, we write the tests first and then write the functionality, in either case, {testthat} makes it pretty seamless.

As a quick refresher, let's look at how to write some basic unit tests in R; for brevity we won't follow a strict TDD workflow - we'll test a couple of functions that we've inherited - `is_even()` and `is_odd()`.

> **Tip**
>
> The `is_odd()` function calls the `is_even()` function, so it makes sense to test the `is_even()` function first. Once we've tested the implementation of `is_even()`, we only need to test the additional logic introduced by `is_odd()`.

12.1 Testable Functions

```
is_even <- function(number) {

  stopifnot(is.numeric(number), number != 0)

  return(number %% 2 == 0)
}
```

```r
is_odd <- function(number) {
  stopifnot(is.numeric(number), number != 0)

  return(!is_even(number))
}
```

Our two functions should take a number as an input, and check that the number isn't zero. `is_even` should return TRUE if the number is divisible by 2 and FALSE otherwise. `is_odd` calls `is_even` and then flips the truth value, so that if a number isn't even, and it's not 0, it's odd.

It clearly makes sense to test `is_even` first, because `is_odd` depends on it. So what should we test? Echoing Einstein's famous words on simplicity, tests should test everything the function does, nothing more and nothing less. Obviously we should check that our input validation is working, if we input 0 do we get an error, the same if we input a non-numeric. Then we should check a few return values, some that the function should return FALSE to and some that the function should return TRUE to. And then there's a slightly less obvious test - our function has one argument and that argument is mandatory, i.e. it has no default value; if we've made this decision we should have made it for a reason, so we should test the function does error if no value is set.

Reminder that unit tests should:

- **be simple**: *tests are not the place to show off what you can do, you should be able to understand at a glance what's being tested and how the test works i.e. favour writing each test out rather than wrapping a bunch in a `vapply()` or a `map()`.*
- **be lightweight**: *you're usually going to want to write hundreds of them for a package and check them often. Each test should run in fractions of a second, a heavy test suite won't be used and so becomes self-defeating*
- **be self-contained**: *don't pass data around between tests, each test_that block should be able to start and terminate in isolation*
- **be informative**: *they should provide helpful error messages when they fail, so that you know precisely which part of your code's logic is broken and where*
- **be comprehensive**: *if your code should do something, write a test to show it does*
- **look both ways**: *if your code shouldn't do something, write a test to show it doesn't*

Writing tests may at times feel cumbersome, but only at the beginning. Once you've got a good test suite up development becomes more enjoyable - less anxiety associated with each change you make or feature you implement - and faster (trust!). You should often feel like you're insulting your own intelligence and that of your colleagues' by writing such a simple test "Well duh, of course it does that…"

12.2 Our First Tests

There are a number of other, more specific tests for more advanced users, but let's stick to expect_error, expect_true, and expect_false for now.

```
library(testthat)
test_that("is_even has an argument called number and it requires an input", {

  expect_true(names(formals(is_even)) == "number")
  expect_error(is_even(),
               regexp = 'argument "number" is missing')
})
```

We're going to make a change to is_even, to show that these tests can fail if the underlying logic of is_even changes resulting in changes in the function's behaviour (this isn't necessary except for explanatory purposes).

```
is_even_inputs <- function() {
  test_that("is_even has an argument called number and it requires an input", {

    expect_true(
      names(formals(is_even)) == "number")

    expect_error(
      is_even(),
      regexp = 'argument "number" is missing'
    )
  })
}


is_even_inputs()
```

Ok, so the test passes. But what if I want to change the input is_even takes to 'x' which is a more common input?

```
is_even <- is_even <- function(x) {

  stopifnot(is.numeric(x), x != 0)

  return(x %% 2 == 0)
```

```
}

is_even_inputs()
```

We see that we get a test failure: – Failure: is_even has an argument called number and it requires an input —– names(formals(is_even)) == "number" is not TRUE

This is exactly what we wanted. We wrote a function, wrote some tests, changed the function's behaviour and then running our tests told us that we'd altered the function's behaviour. At this point we should either fix our function - if indeed we broke it - or update our tests. We'll fix the function as the tests are still doing what we want them to. Then we'll check our old tests still pass.

```
is_even <- function(number) {

  stopifnot(is.numeric(number), number != 0)

  return(number %% 2 == 0)
}

is_even_inputs()
```

Ok, so let's carry on with testing the function. We'll establish that our function doesn't take 0 as an input, and that if we feed it a string, or a string that could be coerced into a numeric that the function errors. This last one might seem like a funny test, but we haven't explicitly asked our function to coerce its inputs, so we should check that it does not.

```
test_that("is_even errors if given a non-numeric input, or 0 as an input", {
  expect_error(is_even(0),
               regexp = stringr::fixed('number != 0'))

  expect_error(
    is_even("string"),
    regexp = "is\\.numeric"
  )

  expect_error(
    is_even("10"),
    regexp = "is\\.numeric"
  )
})
```

And then finally we can test that the return values are what we expect:

```
test_that("is_even returns a logical, and that logical is TRUE if given an even input, and FA
  expect_true(
    inherits(is_even(10), "logical")
  )

  expect_true(
    inherits(is_even(9), "logical")
  )

  expect_true(
    is_even(10) == TRUE
  )
  expect_false(
    is_even(9) == TRUE
    )

  #and another value, just to be sure...
  expect_true(
    is_even(10002) == TRUE
  )
})
```

12.3 Refactoring is_odd

We've now tested that our is_even function does what it should, and doesn't do what it shouldn't. We could add more tests, like what happens if we input a data frame as number, or a factor? Or if 8938957 and 23665 are odds, but we feel quite confident that our current cases take care of those.

We haven't tested is_odd yet, but let's take another look at our function definitions and see if we can't simplify the logic somehwat.

```
is_even <- function(number) {

  stopifnot(is.numeric(number), number != 0)

  return(number %% 2 == 0)
}
```

```
is_odd <- function(number) {
  stopifnot(is.numeric(number), number != 0)

  return(!is_even(number))
}
```

We've written a pretty lightweight and comprehensive test suite for is_even, so do we just go ahead and write the same tests for is_odd? We don't really need to, because is_odd calls is_even anyway. So let's simplify is_odd:

```
is_odd <- function(number) {
  return(!is_even(number))
}
```

Informally test a few values:

```
is_odd("string")
is_odd(0)
```

So we can see that is_odd is producing the errors we would expect it to, because the logic is cemented in is_even. Our tests for is_odd don't *really* need to duplicate this logic, so we could test one each odd-signalling end digit, and each even-signalling end digit.

```
test_that("is_odd returns TRUE for odd numbers and FALSE for even numbers", {

  expect_true(is_odd(11))
  expect_true(is_odd(333))
  expect_true(is_odd(555))
  expect_true(is_odd(37))
  expect_true(is_odd(49))


  expect_false(is_odd(10))
  expect_false(is_odd(4))
  expect_false(is_odd(638))
  expect_false(is_odd(132))
  expect_false(is_odd(666))
})
```

It's overkill to do this, but there's an important point to be made. You might look at this and think 'shouldn't I just apply a list of numbers, rather than write each test out, to avoid duplication?'

```
test_that("is_odd returns TRUE for odd numbers and FALSE for even numbers", {
  odds <- list(11, 333, 555, 37, 49)
  lapply(odds, function(odd) {
    expect_true(is_odd(odd))
  })

  evens <- list(10, 4, 638, 132, 666)
  lapply(evens, function(even){
    expect_false(is_odd(even))
  })
})
```

12.4 Keep it Simple, Stupid

Whilst this is generally good practice, it's not ideal in the case of testing because when a test fails, our error messages are less informative. For brevity we'll add an odd value to our evens list, and apply that list over our tests:

```
test_that("is_odd returns FALSE when given even inputs",{
  evens <- list(10, 4, 638, 132, 666, 17)
  lapply(evens, function(even){
    expect_false(is_odd(even))
  })
})
```

We see that the error message we get back doesn't tell us which of our inputs failed, just that we expected a FALSE and we got a TRUE, somewhere. In this case it's pretty obvious, but there are times when testing things like shiny UI components where it's tempting to put all the UI tags into a list of tags and l/vapply them into an expect function to keep the testing code concise and avoid duplication. However, we want our tests to be informative more than we want them to adhere to Do Not Repeat Yourself principles.

13 Testing Shiny apps

Testing feels pretty straightforward for R packages with {testthat} but it was not built with Shiny in mind. Shiny introduces reactive programming to R users, and it's not self-evident how to test reactive components and applications via {testthat}'s traditional testing approach. In fact, when I sat down to start testing Shiny apps, I realised that not only could I not see how to do it, I didn't know how to articulate why I couldn't `just do it`. I stared at the screen for a while with that unpleasant sense of 'I don't know what I'm doing', looked at a few help pages, and eventually went back to building out more features (don't do this!).

Let's steal a basic shiny app from the `sidebarLayout` documentation. From the code it's pretty clear that we'll have a one page app, with a sidebar layout. In the sidebar we'll have a slider input which allows us to select a number of observations and then in the main panel we'll output a histogram. The server then reacts to changes in the slider's input, and generates a new histogram each time.

```
library(shiny)

# Define UI
ui <- fluidPage(

  # Application title
  titlePanel("Hello Shiny!"),
  sidebarLayout(
    # Sidebar with a slider input
    sidebarPanel(
      sliderInput("obs",
                  "Number of observations:",
                  min = 0,
                  max = 1000,
                  value = 500)
    ),
    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
```

```
)

# Server logic
server <- function(input, output, session) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
if (interactive()) {
# Complete app with UI and server components
shinyApp(ui, server)
}
```

13.1 What's the problem?

Ideally we want to test the three main components - the UI, the server, and the call to combine the two. The first obstacle is the UI, it's not a function like we're used to testing. And we don't need to write a test to check that it's still not a function going forwards, we let the Shiny developers write their own tests.

```
inherits(ui, "function")
```

I think it's quite common to build Shiny apps without really knowing what a shiny.tag.list is, and that's what a UI is.

```
S3Class(ui)
```

And then there's the slightly puzzling unnamed list which has 4 elements

```
length(ui); names(ui)
```

What are the elements, and what kinds of test can we run on them?

```
ui[[1]]
ui[[2]]
ui[2]
ui[3]
class(ui[4][[1]][[1]])
```

We'll come back to this shortly.

The next problem is the server object - which is a function - but a slightly esoteric one.

```
S3Class(server)
```

It takes three mandatory arguments - input, output, and session. The input argument is quite transparent - we use it to access inputs all the time when building Shiny apps, and similar for outputs with the output$ object. As we can index them with $ we think they're probably named lists of some description. But session is a little more opaque.

```
formals(server)
```

So we know that if we want to test the server function we'll need to add input, output, and session but we don't really know what we should add there. Like the UI, we'll come back to this shortly.

So the shinyApp function, which has a more familiar look about it. It takes our ui and server as inputs, and builds the Shiny app for us. Source code:

```
function (ui, server, onStart = NULL, options = list(), uiPattern = "/",
    enableBookmarking = NULL)
{
    if (!is.function(server)) {
        stop("`server` must be a function", call. = FALSE)
    }
    uiPattern <- sprintf("^%s$", uiPattern)
    httpHandler <- uiHttpHandler(ui, uiPattern)
    serverFuncSource <- function() {
        server
    }
    if (!is.null(enableBookmarking)) {
        bookmarkStore <- match.arg(enableBookmarking, c("url",
            "server", "disable"))
        enableBookmarking(bookmarkStore)
    }
    appOptions <- captureAppOptions()
    structure(list(httpHandler = httpHandler, serverFuncSource = serverFuncSource,
        onStart = onStart, options = options, appOptions = appOptions),
        class = "shiny.appobj")
}
```

There's some input validation,

inputs is a list of reactiveValues, output is a list of some values too. Session is… a bit different. And how do we access it programmatically?

Later - refactor to have min > 0, as why would you want 0 breaks and to allow the erro?

14 Testing a Golem Module

There is another layer of complexity if we build our apps with frameworks like {golem}. For the rest of this post, we'll assume some familiarity with {golem} and its modules.

In my case, the modules that I want to test take reactiveValues from other modules, or reactive objects, such as reactive data frames from other modules. This presents a barrier to testing, as in a general R or testthat session, we're not in a reactive context.

Now - I'm pretty sure I first read this in `Mastering Shiny by Hadley Wickham` - but it's important to remember that in R, virtually everything is a function, and reactives are no different. This means that we can mimic the behaviour of a reactive, by passing in a function to a module.

To make it more concrete, I have a module's server function which takes an id and a data frame. The function then calls the moduleServer function, which takes the id from my_module_server, and a server function as an input.

```
my_module_server <- function(id, highlighted_dataframe) {
  moduleServer(id,
               function(input, output, session) # This is what we'd usually have as our serve
  ) {

  }
}
```

At the moment the module doesn't actually do anything, but we have a skeleton in place and we can see that when we call `my_module_server` we have to provide an input for id and highlighted_dataframe.

Let's add some real logic, so that our module creates a reactive object, which updates whenever there's a change in our highlighted_dataframe input, or the updatePlotButton is pressed, and needs to have an x column in the highlighted_dataframe, plus an input set for topN and width + height. This module is a bit more complex, but still less complex than many modules will be.

```
my_module_server <- function(id, highlighted_dataframe) {
  moduleServer(id, function(input, output, session)) {

    reactive_plot <- shiny::eventReactive( c(highlighted_dataframe(), input$updatePlotButton)
      module_plot <- highlighted_dataframe() %>%
        make_module_plot(
          x_var = x,
          top_n = input$topN
        )
      return(module_plot)
    }

    output$modulePlot <- shiny::renderPlot({
      reactive_plot()
    }, res = 100, width = function() input$width, height = function() input$height
    )
    }
  }
}
```

In testing this module we want to know that given the right inputs, a plot is rendered. So how do we go about testing it?

My first pass with testServer was to do something like:

```
testServer(
  app = my_module_server,
  args = list(),
  expr = {
    ns <- session$ns

    #Check input isn't set
    expect_true(is.null(session$topN))

    #Set input
    session$setInputs(topN = 5)
    expect_true(input$topN == 5)

    #... some other code
  }
)
```

This test passed, as did the other tests that I wrote for the inputs, but then I realised that I could set anything as an input here, and the test would pass

```
testServer(
  app = my_module_server,
  args = list(),
  expr = {
    ns <- session$ns

    #Check input isn't set
    expect_true(is.null(session$topN))

    #Set input
    session$setInputs(topN = 5)
    expect_true(input$topN == 5)

    session$setInputs(shalabadaba = "shalabadooo")
    expect_equal(input$shalabadaba, "shalabadooo")
  }
)
```

So I realised that I wasn't testing what I thought I was, or what I needed to test. So I wanted to get a bit more information about what's actually happening in the testServer, like is there actually a reactive_plot being generated?

```
testServer(
  app = my_module_server,
  args = list(),
  expr = {
    ns <- session$ns

    print(reactive_plot())
  }
)
```

So now I get the error that 'highlighted_dataframe was missing', which is a mandatory argument for the module server, now we're getting somewhere. Whereas before the tests were passing because they weren't really testing anything, the test is now failing in meaningful ways.

In more familiar R terms, the server was waiting until it had to do anything with reactive_plot before raising an error. So how do we solve it and check that a plot really is being generated?

15 Notes from other resources

15.1 Mastering Shiny - Testing (Chapter 21 presently)

You can use browser() inside testServer to see what's going on with specific values/what your changes do and what will / won't work…

stopifnot(is.reactive(var)) - nice little trick for input validation in modules, e.g. for highlighted_dataframe()

testServer - Unlike the real world, time does not advance automatically. So if you want to test code that relies on reactiveTimer() or invalidateLater(), you'll need to manually advance time by calling session$elapse(millis = 300).

testServer() ignores UI. That means inputs don't get default values, and no JavaScript works. Most importantly this means that you can't test the update* functions, because they work by sending JavaScript to the browser to simulates user interactions. You'll require the next technique to test such code.

Wrap testServer in test_that

15.2 Shiny App Packages - Testing (Section 3)

Testing the UI

```
mod_bigram_network_ui(id = "test")
```

15.3 Gotchas + Reminders

Browse[1]> class(bigram_reactive) [1] "reactive.event" "reactiveExpr" "reactive" "function" Browse[1]> x <- bigram_reactive() Browse[1]> class(x) [1] "ggraph" "gg" "ggplot"

bigram_reactive is the reactive expression, untriggered. bigram_reactive() is the actual ggraph/gg/ggplot object now triggered. Always good to remind onself of this and what that means when interacting with the objects at various points of the SWD process.

Can use `ui <- mod_bigram_network_ui(id = "test")` and type ui to see all of the shiny tags, and then type ui[[1]] to render the UI in a viewer object, maybe easier than end app, run_app() -> click to app.

15.4 Testing interaction with nested modules

the mod_group_vol_time_server gets its filtered data out of the mod_daterange_input_server. This presents a challenge because each module is namespaced, so we can't just `setInputs(dateRange = list(as.Date("2023-01-03"), as.Date("2023-01-09"))` because we'd be setting the value of dateRange inside the wrong namespace - the namespace of mod_group_vol_time_server.

This means when we try to access the `group_date_range_vot$over_time_data()` to generate our groupued volume over time char, we get an error that the dateRange isn't set. So with the help of `ui <- mod_group_vol_time_ui(id = "test")` we can look for the correct input to set `dateRange` to, which is `dateRangeGroupVol-dateRange`. We've unearthed a general truth about nested modules. Our parent module is dateRangeGroupVol, our child is dateRange, so we join the two with `dateRangeGroupVol-dateRange`, if dateRange had a child module called dateRangeChild, we'd join the three with `dateRangeGroupVol-dateRange-dateRangeChild`!

```
session$setInputs(
  # ns("dateRange") = list(as.Date("2023-01-03"), as.Date("2023-01-09")),
  dateBreak = "day",
  height = 600,
  width = 400,
  nrow = 2,
  #So to pass stuff into the modules that need them we can pre-prepend the namespace with sy
  `dateRangeGroupVol-dateRange` = list(as.Date("2023-01-03"), as.Date("2023-01-09"))
)
```

15.5 Emulating a reactive with the return values specified:

because in our module we try to: label_ids <- as.numeric(selected_range()$key)

We can't just send in selected_range as c(1, 2, 3). We need to send it in as an object which imitates a reactive, with a return value of list(key = c(...)) so that when we call it in the module later on, we get the current value, and we can access the key. Tricky.

selected_range = function(){ return(list(key = c(1, 2, 3))) }

If wanting to change the length of the generate_dummy_data for whatever reason: function(){return(generate_dummy_data(length = 20))}

Part V

Tips and Tricks

16 Coding best practices

First, make sure you've read the the Project Management document for general tips on setting up an RStudio project and working with file paths.

> **Note**
>
> To avoid being cumbersome, we'll use 'notebook' to refer to the set of interactive software in which data science usually takes place. They'll tend to end with one of the following extensions: .md, .Rmd, .qmd, or .ipynb.

16.1 Why are we here?

At SAMY, code is the language of both our research and our development, so it pays to invest in your coding abilities. There are many great (and many terrible) resources on learning how to code. This document will focus on practical tips on how to structure your code to reduce cognitive strain and do the best work you can.

Let's be clear about what coding is: coding is thinking not typing, so good coding is simply good thinking and arranging our code well will help us to think better.

16.2 Reproducible Analyses

Above everything else, notebooks must be reproducible. What do we mean by reproducible? You and your collaborators should be able to get back to any place in your analysis simply by executing code in the order it occurs in your scripts and notebooks. Hopefully the truth of this statement is self-evident. But if that's the case, why are we talking about it?

For some projects you'll get away with a folder structure which looks something like this:

```
example_folder
+-- code
|   \-- analysis.Rmd
\-- data
    +-- clean_data.csv
```

```
    \-- raw_data.csv
```

However, in weeks-long or even months-long research projects, if you're not careful your project will quickly spiral out of control (see the lovely surprise below for a tame example), your R Environment will begin to store many variables, and you'll begin to pass data objects around between scripts and markdowns in an unstructured way, e.g. you'll reference a variable created inside 'wrangling.Rmd' inside the 'colab_cleaning.Rmd', such that colab_cleaning.Rmd becomes unreproducible.

A lovely surprise

```
example_folder_complex
+-- code
|   +-- colab_cleaning.Rmd
|   +-- edited_functions.R
|   +-- images
|   |   \-- outline_image.png
|   +-- initial_analysis.Rmd
|   +-- quick_functions.R
|   +-- topic_modelling.Rmd
|   \-- wrangling.Rmd
\-- data
    +-- clean
    |   +-- all_data_clean.csv
    |   +-- all_data_clean_two.csv
    |   +-- all_data_cleaner.csv
    |   +-- data_topics_clean.csv
    |   \-- data_topics_newest.csv
    \-- raw
        +-- sprinklr_export_1.xlsx
        +-- sprinklr_export_10.xlsx
        +-- sprinklr_export_11.xlsx
        +-- sprinklr_export_12.xlsx
        +-- sprinklr_export_13.xlsx
        +-- sprinklr_export_14.xlsx
        +-- sprinklr_export_15.xlsx
        +-- sprinklr_export_16.xlsx
        +-- sprinklr_export_17.xlsx
        +-- sprinklr_export_18.xlsx
        +-- sprinklr_export_19.xlsx
        +-- sprinklr_export_2.xlsx
        +-- sprinklr_export_20.xlsx
        +-- sprinklr_export_21.xlsx
```

```
+-- sprinklr_export_22.xlsx
+-- sprinklr_export_23.xlsx
+-- sprinklr_export_24.xlsx
+-- sprinklr_export_25.xlsx
+-- sprinklr_export_26.xlsx
+-- sprinklr_export_27.xlsx
+-- sprinklr_export_28.xlsx
+-- sprinklr_export_29.xlsx
+-- sprinklr_export_3.xlsx
+-- sprinklr_export_30.xlsx
+-- sprinklr_export_4.xlsx
+-- sprinklr_export_5.xlsx
+-- sprinklr_export_6.xlsx
+-- sprinklr_export_7.xlsx
+-- sprinklr_export_8.xlsx
\-- sprinklr_export_9.xlsx
```

16.2.1 Literate Programming

> "a script, notebook, or computational document that contains an explanation of the program logic in a natural language (e.g. English or Mandarin), interspersed with snippets of macros and source code, which can be compiled and rerun. You can think of it as an executable paper!"

Notebooks have become the de factor vehicles for Literate Programming and reproducible research. They allow you to couple your code, data, visualisations, interpretations and analysis. You can and should use the knit/render buttons regularly (found in the RStudio IDE) to keep track of whether your code is reproducible or not - follow the error messages to ensure reproducibility.

- ☐ Have I turned off the restore .RData setting in tools –> global options?
- ☐ Have I separated raw data and clean data?
- ☐ Have I recorded in code my data cleaning & transformation steps?
- ☐ Do my markdowns and notebooks render?
- ☐ Am I using relative or absolute filepaths within my scripts & notebooks?
- [ ]
- [ ]

16.3 On flow and focus

Most of us cannot do our best work on the most difficult challenges for 8 hours per day. In fact, conservative estimates suggest we have 2-3 hours per day, or 4 hours on a good day, where we can work at maximum productivity on challenging tasks. Knowing that about ourselves, we should proactively introduce periods of high **and** low intensity to our days.

In periods of high intensity we'll be problem solving - inspecting our data, selecting cleaning steps, running small scale experiments on our data: 'What happens if I…' and recording and interpreting the results. When the task is at the correct difficulty, you'll naturally fall into a flow state. Try your best to prevent interruptions during this time. Protect your focus - don't check your work emails, turn Slack off etc.

Whilst these high-intensity periods are rewarding and hyper-productive, at the other end there is often a messy notebook or some questionable coding practices. Allocate time each and every day to revisit the code, add supporting comments, write assertions and tests, rename variables to be more descriptive, tidy up unused data artefacts, study your visualisations to understand what the data can really tell you etc. or anything else you can do to let your brain rest, recharge and come back stronger tomorrow. You'll sometimes feel like you don't have time to do these things, but it's quite the opposite - you don't have time not to do them.

16.4 On managing complexity

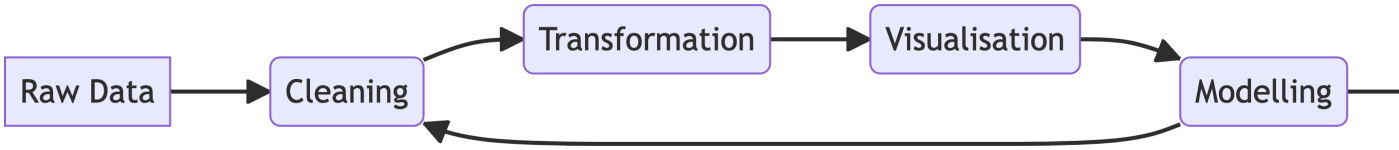
> "…let's think of code complexity as how difficult code is to reason about and work with."

There are many heuristics for measuring code complexity, the most basic being 'lines of code' which is closely linked to 'vertical complexity' - the more code we have the longer our scripts and markdowns will be, the harder it is to see all of the relevant code at any one time, the more strain we put on our working memories. A naive strategy for reducing complexity is to reduce lines of code. But if we reduce the number of lines of code by introducing deeply nested function calls, the code becomes more complex not less as the number of lines decreases.

As a rough definition, let's think of code complexity as 'how difficult code is to reason about and work with.' A good test of code complexity is how long it takes future you to remember what each line, or chunk, of code is for.

We'll now explore some tools and heuristics for fighting complexity in our code.

16.5 On navigation



Let's go out on a limb and say that the data science workflow is **never** linear, you will always move back and forth between cleanin data, inspecting it, and modelling it. Structuring your projects and notebooks with this in mind will save many headaches.

16.5.1 Readme

For each project, add a`README.md` or `README.Rmd`, here you can outline what and who the project is for and guide people to notebooks, data artefacts, and any important resources. You may find it useful to maintain a to-do list here, or provide high-level findings - it's really up to you, just keep your audience in mind.

16.5.2 Section Titles

Section titles help order your thoughts - when done well they let you see the big picture of your document. They will also help your collaborators to navigate and understand your document, and they'll function as HTML headers in your rendered documents. When in the RStudio IDE the outline tab allows click-to-navigate with your section titles.

> Tip
>
> Set the toc-depth: in your quarto yaml to control how many degrees of nesting are shown in your rendered document's table of contents.

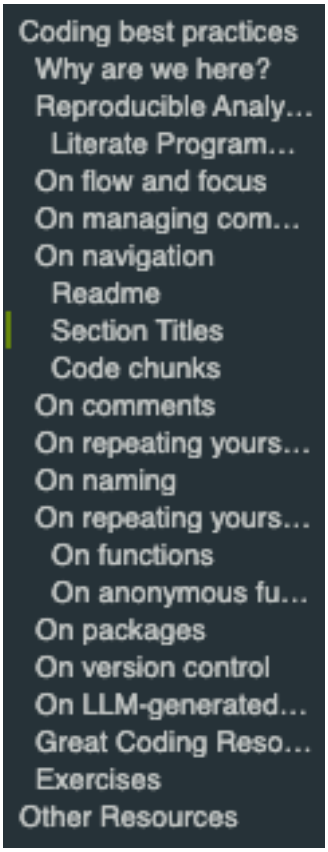


Figure 16.1: Rstudio Outline

16.5.3 Code chunks

You wrote the code in the chunk. So you know what it does, or at least you should. However, when rendering your document (which you should do regularly) it's handy to have named chunks so that you know precisely which chunk is taking a long time to render, or has a problem. Furthermore, that 8-line pipe inside the chunk might not be as easy to understand at a glance in the future, and it certainly won't be for your collaborators. It's much easier to understand what a descriptively named chunk is doing than 8 piped function calls.

16.6 On comments

When following the literate programming paradigm, coding comments (`# comment...`) should be included in code chunks with `echo = False` unless you explicitly want your audience to see the code and the comments - save the markdown text for what your audience needs to see.

Generally code comments should be used sparingly, if you find yourself needing a lot of comments it's a sign the code is too complex, consider re-factoring or abstracting (more on abstractions later).

16.7 On repeating yourself #1 - Variables

Storing code in multiple places tends to be a liability - if you want to make changes to that piece of code, you have to do it multiple times. More importantly than the time lost making the changes, you need to remember that the code has been duplicated and where all the copies are.

Without variables coding would be 'nasty, brutish and short long.'. It's difficult to find the Goldilocks zone between 'more variables than I can possibly name' and 'YOLO the project title is typed out 36 times'.

Magrittr's pipe operator (%>% or command + shift + m) can save you from having to create too many variables. It would be quite ugly if we had to always code like this:

```
mpg_horesepower_bar_chart <- ggplot(mtcars, aes(x = mpg, y = hp))

mpg_horesepower_bar_chart  <- mpg_horesepower_bar_chart  + geom_point()

mpg_horesepower_bar_chart  <- mpg_horesepower_bar_chart  + labs(title = "666 - Peaks & Pits -

mpg_horesepower_bar_chart
```

Instead of this:

```
mtcars %>%
  ggplot(aes(x = mpg, y = hp)) +
  geom_point() +
  labs(title = "666 - Peaks & Pits - Xbox Horsepower vs Miles per Gallon")
```

Place strings you'll use a lot in variables at the top of your notebook, and then use the paste function, rather than cmd + c, to use the contents of the variable where necessary. This way, when you need to change the title of the project you won't have to mess around with cmd + f or manually change each title for every plot.

```r
project_title <- "666 - Peaks & Pits - Xbox:"

mtcars %>%
  ggplot(aes(x = mpg, y = hp)) +
  geom_point() +
  labs(title = paste0(project_title, " Horsepower vs Miles per Gallon"))
```

Give your variables descriptive names and use your IDE's tab completion to help you access long names.

Let's say you're creating a data frame that you're not sure you'll need. Assume you will need it and delete after if not, don't fall into the trap of naming things poorly

tmp_df

screen_name_counts

16.8 On naming

The primary objects for which naming is important are variables, functions, code chunks, section titles, and files. Give each of these clear names which describe precisely what they do or why they are there.

16.9 On repeating yourself #2 - Abstractions

Do Not Repeat Yourself, so the adage goes. But some repetition is natural, desirable, and harmless whereas attempts to avoid all repetition can be the opposite. As a rule-of-thumb, if you write the same piece of code three times you should consider creating an abstraction.

Reasonable people disagree on the precise definition of 'abstraction' when it comes to coding & programming. For our needs, we'll think about it as simplifying code by hiding some complexity. A good abstraction helps us to focus only on the important details, a bad abstraction hides important details from us.

The main tools for creating abstractions are:

- Functions
- Classes
- Modules
- Packages

We'll focus on functions and packages.

16.9.1 On functions

Make them! There are *lots* of reasons to write your own functions and make your code more readable and re-usable. We can't hope to cover them all here, but we want to impress their importance. Writing functions will help you think better about your code and understand it on a deeper level, as well as making it easier to read, understand and maintain.

For a more comprehensive resource, check in with the R4DS functions section

Also see the Tidyverse Design Guide for stellar advice on building functions for Tidyverse functions.

16.9.2 On anonymous functions

Functions are particularly useful when you want to use iterators like {purrr}'s `map` family of functions or base R's `apply` family. Often these functions are one-time use only so it's not worth giving them a name or defining them explicitly, in which case you can use anonymous functions.

Anonymous functions can be called in three main ways:

1. Using `function()` e.g. `function(x) x + 2` will add 2 to every input
2. Using the new anonymous function notation: `\x x + 2`
3. Using the formula notation e.g. `map(list, ~.x + 2)`

You will see a mixture of these, with 3. being used more often in older code, and 2. in more recent code.

16.10 On packages

Depending on how many functions you've created, how likely you are to repeat the analysis, and how generalisable the elements of your code are, it may be time to create a package.

At first building a package is likely to seem overwhelming and something that 'other people do'. However, in reality the time it takes to create a package reduces rapidly the more you create them. And the benefits for sharing your code with others are considerable. Eventually you'll be able to spin up a new package for personal use in a matter of minutes, over time it will become clear which packages should be developed, left behind, or merged into an existing SAMY package.

Visit the Package Development document for practical tips and guidelines for developing R packages

see also: Package Resources

16.11 On namespaces and function conflicts

R manages functions via namespaces. Depending on the order that you import your packages, you may find a function doesn't behave as expected. For example, the {stats} package has a `filter()` function, but so does {dplyr}. By default R will use the most recently-imported package's namespace to avoid any conflicts, so `filter()` will now refer to {dplyr}'s implementation.

If you're experiencing weirdness with a function, you may want to restart your R session, change the order of your imports to prevent the same weirdness occurring again. However, a more straightforward approach is to use the `package::` notation to explicitly refer to the function you intended to use e.g. `dplyr::filter()` will avoid any potential conflicts and confusion.

16.12 On version control

By default your projects should be stored on Google Drive inside the "data_science_project_work" folder, in the event of disaster (or minor inconvenience) this means your code and data artefacts should be backed up. However, it's still advisable to use a version control system like git - using branches to explore different avenues, or re-factor your code, can be a real headache preventer and efficiency gain.

Aim to commit your code multiple times per day, push to a remote branch (not necessarily main or master) once a day and merge + pull request when a large chunk of work has been finished. Keep your work code and projects in a private repository, add .Rhistory to .gitignore and make sure API keys are stored securely, i.e. not in scripts and notebooks.

16.13 On Managing Dependencies

Applying the Anna Karenina principle to virtual environments:

> "All happy virtual environments are alike; each unhappy virtual environment is unhappy in its own way"

16.13.1 R

The R ecosystem - led by CRAN and posit (formerly RStudio) - does a great job in managing package-level dependencies. It's rare to end up in dependency hell R. However, there is still scope for 'works on my machine' when working with collaborators who have different versions

of a package, e.g. person 1 upgrades their {dplyr} version before person 2, and now that .by argument person 1 has used is breaking code on person 2's machine.

To avoid this, we advise using something like the {renv} package to manage package versions.

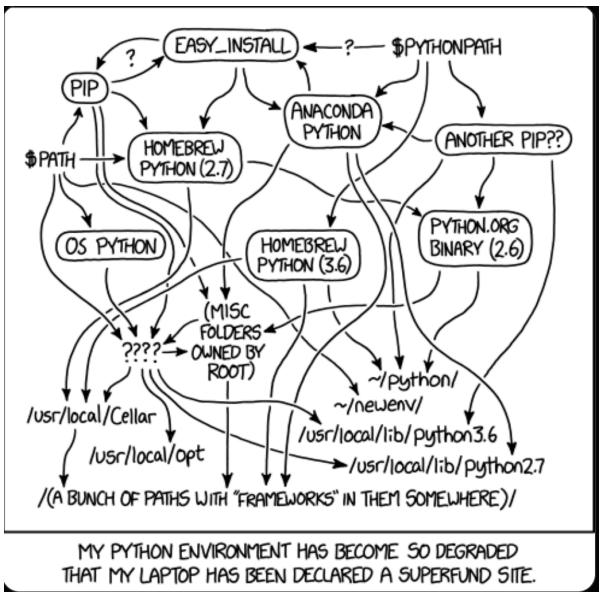
On using renv collaboratively

renv helps us keep track of package & R versions, which makes deployment 10x easier than without it. However, it can get tricky if we're using renv in different ways.
Start a renv project off with`renv::init()`, this will essentially remove all of your packages. You could choose to take your current packages with you, but it's not advised. These packages are linked to your RStudio project when using RStudio, which means other RStudio projects will have your current packages if they're not themselves using renv.
`renv::init()` creates a renv lockfile, you'll need to keep this up to date as you work through the project - especially important if collaborating with other people on a project that uses renv. Once installed inside your local project, you can add single packages to the lockfile by `renv::record("package_name")`. This is preferable to adding a bunch of packages at a time with `renv::snapshot()` particularly when collaborating. Generally it will be better to give one person control of the lockfile and to communicate about adding packages as and when.
If you're working in a Quarto Doc or an RMarkdownfile to develop things that don't need to pushed to the repo, you can create a `.renvignore` file like .Rbuildignore, .gitignore etc. and add the folder where the markdowns sit to make sure renv doesn't try to sync itself with the packages you're experimenting with.
At any time you can check your environment is still synced with `renv::status()`, if your project is out of sync, you may want to `renv::clean()`, if you've got a bunch of packages that are like this:

```
The following package(s) are in an inconsistent state:
package       installed recorded used
backports     y         y        n
blob          y         y        n
broom         y         y        n
callr         y         y        n
cellranger    y         y        n
```

Then you'll need to `renv::clean(actions = "unused.packages")`, which should get you in working order. There's a lot more to {renv} when collaborating but these steps will do a lot to keep your environment in sync and allow collaborators to use your code.

16.13.2 Python



Unlike R, it's pretty easy to get into deep, deep trouble when working with Python environments. We advise using miniconda, keeping your base environment completely free of packages, and creating virtual environments for projects or large, but commonly-used and important packages like torch.

> **Tip**
>
> Just remember to activate the correct environment every time you need to install a package!

16.14 On LLM-generated code

GitHub Copilot, ChatGPT, Claude and other LLM-based code generators can be extremely useful, but they are a double-edged sword and should be used responsibly. If you find yourself relying on code you don't understand, or couldn't re-build yourself, you're going to run in to trouble somewhere down the line. You have the time and space to learn things deeply here, so do read the docs, do reference textbooks, and do ask for help internally before relying on LLM-generated code which often **looks right** but is outdated or subtly incorrect/buggy.

> **Tip**
>
> You're here because you can problem solve and pick up new skills when you need them - don't be afraid to spend extra time understanding a concept or a library.

16.15 Great Coding Resources

- Google SWE Book
- Hands on programming with R
- Reproducible Analysis, Jenny Bryan

16.16 Exercises

In your own words, summarise what makes an analysis reproducible.

Write a line in favour and against the claim 'Code is an asset not a liability.'

Set up a private github repo on a project inside data_science_project_work/internal_projects and create a new branch then commit, push and pull request a change.

Add your own best practices to this document!

17 Other Resources

- cognitive strain
- coding is thinking not typing
- Google Code Health

18 Data Cleaning

When we receive a dataset from the Insight team, the first step that we must take involves cleaning and pre-processing the text data.

Broadly, this data cleaning for unstructured textual data can be categorised into levels of cleaning:

- dataset-level
- document-level

18.1 Dataset-level cleaning

Goal: Ensure the dataset as a whole is relevant and of high quality

The main steps that we take for this level of cleaning is *spam removal*, *uninformative content removal* and *deduplication*

18.1.1 Spam Removal

We use the term "spam" quite loosely in our data pre-processing workflows. Whilst the strict definition of "spam" could be something like "unsolicited, repetitive, unwanted content", we can think of it more broadly any post that displays irregular posting patterns or is not going to provide analytical value to our research project.

18.1.1.1 Hashtag filtering

There are multiple ways we can identify spam to remove it. The simplest is perhaps something like hashtag spamming, where an excessive number of hashtags, often unrelated to the content of the post, can be indicative of spam.

We can identify posts like this by counting the number of hashtags, and then filtering out posts that reach a certain (subjective) threshold.

```
cleaned_data <- data %>%
  mutate(extracted_hashtags = str_extract_all(message_column, "#\\S+"),
         number_of_hashtags = lengths(extracted_hashtags)) %>%
  filter(number_of_hashtags < 5)
```

In the example above we have set the threshold to be 5 (so any post that has 5 or more hashtags will be removed), however whilst this is a valid starting point, it is highly recommend to treat each dataset uniquely in determining which threshold to use.

18.1.1.2 Spam-grams

Often-times spam can be identified by repetitive posting of the same post, or very similar posts, over a short period of time.

We can identify these posts by breaking down posts into n-grams, and counting up the number of posts that contain each n-gram. For example, we might find lots of posts with the 6-gram "Click this link for amazing deals", which we would want to be removed.

To do this, we can unnest our text data into n-grams (where we decide what value of n we want), count the number of times each n-gram appears in the data, and filter out any post that contains an n-gram above this filtering threshold.

Thankfully, we have a function within the `LimpiaR` package called `limpiar_spam_grams()` which aids us with this task massively. With this function, we can specify the value of n we want and the minimum number of times an n-gram should occur to be removed. We are then able to inspect the different n-grams that are removed by the function (and their corresponding post) optionally changing the function inputs if we need to be more strict or conservative with our spam removal.

```
spam_grams <- data %>%
  limpiar_spam_grams(text_var = message_column,
                     n_gram = 6,
                     min_freq = 6)

# see remove spam_grams
spam_grams %>%
  pluck("spam_grams")

# see deleted posts
spam_grams %>%
  pluck("deleted")
```

```
# save 'clean' posts
clean_data <- spam_grams %>%
  pluck("data")
```

18.1.1.3 Filter by post length

Depending on the specific research question or analysis we will be performing, not all posts are equal in their analytical potential. For example, if we are investigating what specific features contribute to the emotional association of a product with a specific audience, a short post like "I love product" (three words) won't provide the level of detail required to answer the question.

While there is no strict rule for overcoming this, we can use a simple heuristic for post length to determine the minimum size a post needs to be before it is considered informative. For instance, a post like "I love product, the features x and y excite me so much" (12 words) is much more informative than the previous example. We might then decide that any post containing fewer than 10 words (or perhaps 25 characters) can be removed from downstream analysis.

On the other end of the spectrum, exceedingly long posts can also be problematic. These long posts might contain a lot of irrelevant information, which could dilute our ability to extract the core information we need. Additionally, long posts might be too lengthy for certain pipelines. Many embedding models, for example, have a maximum token length and will truncate posts that are longer than this, meaning we could lose valuable information if it appears at the end of the post. Also, from a practical perspective, longer posts take more time to analyse and require more cognitive effort to read, especially if we need to manually identify useful content (e.g. find suitable verbatims).

```
# Remove posts with fewer than 10 words
cleaned_data <- data %>%
  filter(str_count(message_column, "\\w+") >= 10)

# Remove posts with fewer than 25 characters and more than 2500 characters
cleaned_data <- data %>%
  filter(str_length(message_column) >= 25 & str_length(message_column) <= 2500)
```

18.1.2 Deduplication

While removing spam often addresses repeated content, it's also important to handle cases of exact duplicates within our dataset. Deduplication focuses on eliminating instances where entire data points, including all their attributes, are repeated.

A duplicated data point will not only have the same message_column content but also identical values in every other column (e.g., universal_message_id, created_time, permalink). This is different from spam posts, which may repeat the same message but will differ in attributes like universal_message_id and created_time.

Although the limpiar_spam_grams() function can help identify spam through frequent n-grams, it might not catch these exact duplicates if they occur infrequently. Therefore, it is essential to use a deduplication step to ensure we are not analysing redundant data.

To remove duplicates, we can use the `distinct()` function from the `dplyr` package, ensuring that we retain only unique values of `universal_message_id`. This step guarantees that each post is represented only once in our dataset.

```
data_no_duplicates <- data %>%
  distinct(universal_message_id, .keep_all = TRUE)
```

18.2 Document-level cleaning

Goal: Prepare each individual document (post) for text analysis.

At a document-level (or individual post level), the steps that we take are more small scale. The necessity to perform each cleaning step will depend on the downstream analysis being performed, but in general the different steps that we can undertake are:

18.2.1 Remove punctuation

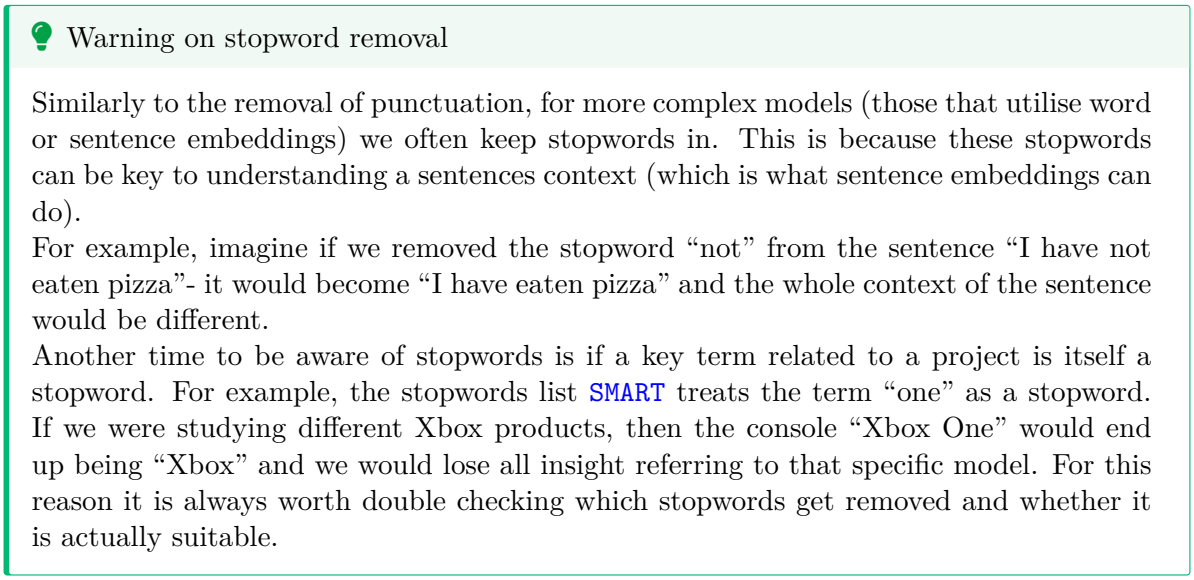
Often times we will want punctuation to be removed before performing an analysis because they *tend* to not be useful for text analysis. This is particularly the case with more 'traditional' text analytics, where an algorithm will assign punctuation marks a unique numeric identify just like a word. By removing punctuation we create a cleaner dataset by reducing noise.

> Warning on punctuation
>
> For more complex models, such as those that utilise word or sentence embeddings, we often keep punctuation in. This is because punctuation is key to understanding a sentences context (which is what sentence embeddings can do).
> For example, there is a big difference between the sentences "Let's eat, Grandpa" and "Let's eat Grandpa", which is lost if we remove punctuation.

18.2.2 Remove stopwords

Stopwords are extremely common words such as "and," "the," and "is" that often do not carry significant meaning. In text analysis, these words are typically filtered out to improve the efficiency of text analytical models by reducing the volume of non-essential words.

Removing stopwords is particularly useful in our projects for when we are visualising words, such as a bigram network or a WLO plot, as it is more effective if precious informative space on the plots is not occupied by these uninformative terms.



18.2.3 Lowercase text

Converting all text to lowercase standardises the text data, making it uniform. This helps in treating words like "Data" and "data" as the same word, and is especially useful when an analysis requires an understanding of the frequency of a term (we rarely want to count "Data" and "data" as two different things) such as bigram networks.

18.2.4 Remove mentions

Mentions (e.g., `@username`) are specific to social media platforms and often do not carry significant meaning for text analysis, and in fact may be confuse downstream analyses. For example, if there was a username called `@I_love_chocolate`, upon punctuation remove this might end up confusing a sentiment algorithm. Removing mentions therefore helps in focusing on the actual content of the text.

Retaining mentions, sometimes

We often perform analyses that involve network analyses. For these, we need to have information of usernames because they appear when users are either mentioned or retweeted. In this case we do not want to remove the `@username` completely, but rather we can store this information elsewhere in the dataframe.
However, broadly speaking if the goal is to analyse the content/context of a paste, removing mentions is very much necessary.

18.2.5 Remove URLs

URLs in posts often point to external content and generally do not provide meaningful information for text analysis. Removing URLs helps to clean the text by eliminating these irrelevant elements.

18.2.6 Remove emojis/special characters

Emojis and special characters can add noise to the text data. While they can be useful for certain analyses (like emoji-specific sentiment analysis - though we rarely do this), they are often removed to simplify text and focus on word-based analysis.

18.2.7 Stemming/Lemmatization

Stemming and lemmatization are both techniques used to reduce words to their base or root form and act as a text normalisation technique.

Stemming trims word endings to their most basic form, for example changing “clouds” to “cloud” or “trees” to “tree”. However, sometimes stemming reduces words to a form that doesn’t make total sense such as “little” to “littl” or “histories” to “histori”.

Lemmatization considers the context and grammatical role when normalising words, producing dictionary definition version of words. For example “histories” would become “history”, and “caring” would become “car” (whereas for stemming it would become “car”).

We tend to use lemmatization over stemming- despite it being a bit slower due to a more complex model, the benefit of lemmitization outweighs this. Similar to lowercasing the text, lemmitization is useful when we need to normalise text where having distinct terms like “change”, “changing”, “changes”, and “changed” isn’t necessary and just “change” is suitable.

18.3 Conclusion

Despite all of these different techniques, it is important to remember these are not mutually exclusive, and do not always need to be performed. It may very well be the case where a specific project actually required us to mine through the URLs in social posts to see where users a linking too, or perhaps keeping text as all-caps is important for how a specific brand or product is mentioned online. Whilst we can streamline the cleaning steps by using the `ParseR` function above, it is **always** worth spending time considering the best cleaning steps for each specific part of a project. It is much better spending more time at the beginning of the project getting this right, than realising that the downstream analysis are built on dodgy foundations and the data cleaning step needs to happen again later in the project, rendering intermediate work redundant.

19 Calling APIs

An API (Application Programming Interface) is a mechanism that allows two software components to communicate with each other and share data or functionality. In simple terms, it enables us to send a request to some software, such as a model, and receive information in return. APIs simplify this process by abstracting the underlying complexity, allowing for smooth information exchange.

19.1 Why Do We Use APIs?

As a team, our main use case for APIs is the OpenAI API, which grants us access to the advanced AI models developed by OpenAI, including the GPT (text), DALL-E (image generation), and Whisper (speech-to-text) models. One of the key advantages of using an API instead of downloading and running these models locally (or utilising open-source models) is that it allows us to leverage the computational power and optimisation of the models without needing expensive hardware or vast computational resources.

19.2 OpenAI API Overview

OpenAI's API is a REST (Representational State Transfer) API. Simply put, this allows a **client** (such as our program) to request data or perform actions on a **server** (which hosts the AI models), where it retrieves or manipulates **resources** (e.g., model outputs such as generated text).

19.2.1 How the API Works

OpenAI's API works on the standard HTTP protocol, which structures communication between the client and server. In this system:

1. **Endpoints** are specific paths on the server where the API can be accessed. For example, `/v1/chat/completions` is an endpoint that allows us to send prompts to a GPT model and receive completions.

2. **Requests** are the actions taken by our application. We send requests with specific inputs (like text prompts), and the API processes them.
3. **Responses** are the API's outputs, such as text from a GPT model, an image from DALL-E, or speech-to-text conversions from Whisper.

19.3 Practical Use of OpenAI's API

We use the OpenAI API similarly to other public APIs: sign up for an account, obtain an API key, and use it to make API calls to specific models using HTTP requests.

19.3.1 Step One - Obtain API Key and Authentication

To start using OpenAI's API, you'll need an API key for authentication. Follow these steps:

1. Go to platform.openai.com and create an account using your SHARE email address.
2. Mike will add you to the "SHARE organization" within the platform, allowing you to access the set aside usage credits we have as a company.
3. Then make your way to the api-keys section of the platform and click the green `Create new secret key` in the top corner.

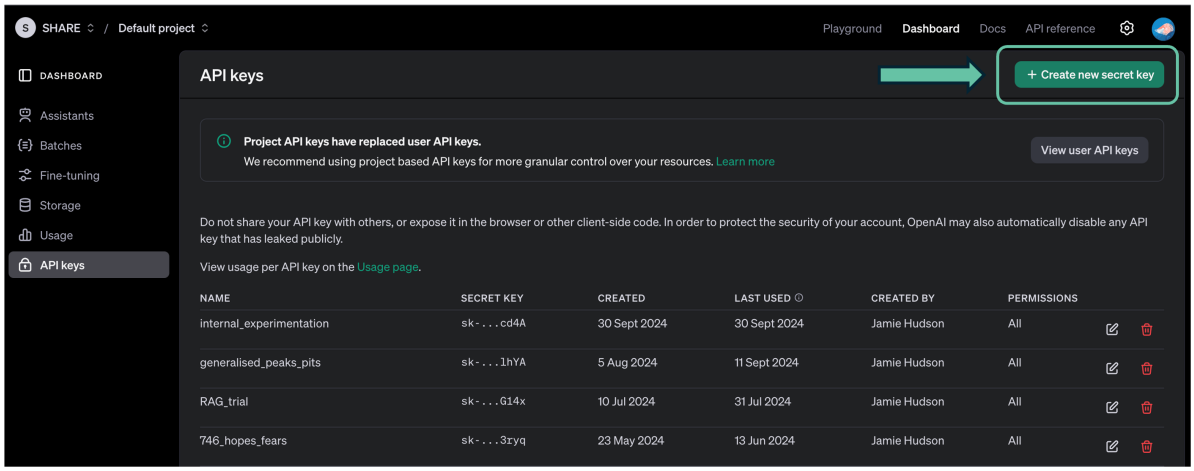


Figure 19.1: Create new secret key by clicking the button in the top right hand corner

4. Rename the key to something useful, such as the name and number of the project that they key will be used for, and keep the OpenAI project as "Default project" and Permissions as "All".

5. You will then be provided with the opportunity to copy the provided API key, this is the one chance you will get to obtain it- after you click off this pop up you won't be able to view the full API key again and you'll need to request a new one. Because of this, make sure you copy the key and add it to this private Google Sheet where the DS team keeps the API Keys. Remember that using the API costs money, so if this key is used by others we risk someone using up all of our API credits! Please see below for some more best practices relating to API key security.

19.3.2 Step Two - Managing API Keys Securely

As outlined above, when working with APIs it's essential to manage our API keys securely. An API key grants access to services, and if exposed, others could misuse it, leading to security breaches, unauthorised usage, or unexpected costs. Here are some key principles to follow:

1. **Never Hard-Code API Keys** Avoid storing API keys directly in your code as hard-coded variables. This exposes them to anyone with access to your codebase.
2. **Use Environment Variables** Store API keys in environment variables to keep them separate from the code. This ensures sensitive data isn't exposed, and it's easier to manage keys across different environments if required (development, production, etc.).
3. **Version Control Precautions** Make sure to add environment files that contain sensitive information (like `.env` or `.Renviron`) to .gitignore so they don't get uploaded to version control systems like GitHub. Exposing API keys in public repositories is a common mistake, and it can be a serious security risk.

Example implementations

RStudio

1. Add API Key to `.Renviron`

Use the `usethis` package to edit the `.Renviron` file where environment variables are stored. Add the API key like this:

```
usethis::edit_r_environ(scope = "project")
```

This will open the `.Renviron` file in your editor. Note that `scope = "project"` scope means that the `.Renviron` file will be created in your specific R project folder. This means the environment variables (like your API key) will only be available when you are working inside that project. It's a good way to keep project-specific configuration separate from other projects.
Then add the following line to store your API key (replace `your-api-key-here` with the actually API key)

```
# Write this within the .Renviron file and save it
OPENAI_API_KEY=your-api-key-here
```

2. Access the API Key in your R Script

You can access the API key in your R scripts using Sys.getenv()

```
api_key <- Sys.getenv("OPENAI_API_KEY")
```

or if you need to call the API key in a function (such as BERTopicR) it could be

```
representation_openai <- bt_representation_openai(fitted_model,
                                                  documents,
                                                  openai_model = "gpt-4o-mini",
                                                  nr_repr_docs = 10,
                                                  chat = TRUE,
                                                  api_key = Sys.getenv("OPENAI_API_KEY"))
```

3. Add `.Renviron` to `.gitignore`

Obviously this is only relevant if you are deploying a repo/project to GitHub, but we can make sure to exclude the `.Renviron` file to our `.gitignore` file

```
# Exclude .Renviron file
.Renviron
```

Python

1. Create a `.env` file

In the root directory of your project, create a .env file. The best way to do this is using command line tools (`touch` and `nano`)
Within the terminal create an empty `.env` file by running

```
touch .env
```

and then edit it by running

```
nano .env
```

and finally within the `nano` editor, type the following to add your API key (replace `your-api-key-here` with the actually API key)

```
OPENAI_API_KEY=your-api-key-here
```

2. Use the `python-dotenv` library

Install `python-dotenv` by running

```
pip install python-dotenv
```

3. Access the API Key in your script

In your Python script, load the `.env` file and access the API key

```
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

# Access the API key
api_key = os.getenv("OPENAI_API_KEY")
```

4. Add `.env` to `.gitignore`

Similar to the RStudio implementation above, add `.env` to your `.gitignore`

```
# Exclude .env file
.env
```

19.3.3 Step Three - Making Requests to the API

To actually make requests to the OpenAI API we use python, and specifically the official OpenAI SDK. You can install it to your python environment simply via pip by running

```
pip install openai
```

The documentation on-line surrounding calling the OpenAI API is extremely extensive and generally good, however the API and underlying models do get updated quite often and this can cause code to become redundant or not act as one may expect. This can be particularly unwelcome when you run a previously working script to ping the API, **get charged**, but don't receive an output that is useful.

The simple way to call the API and obtain a 'human-like response' to a prompt is with this code adapted from the OpenAI API tutorial:

```
from openai import OpenAI
client = OpenAI(api_key = OPENAI_API_KEY)

completion = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Write a short poem about RStudio."}
    ]
)

print(completion.choices[0].message)
```

Don't worry about what everything means, we'll explain this in a bit more detail below. But firstly, one thing to realise is that this code above is effectively the same as going onto the ChatGPT website and typing into the input box "You are a helpful assistant. Write a short poem about RStudio." for the model `gpt-4o-mini`. So effectively this code calls the API once, with an input and receives an output from the model.

19.3.3.1 Chat Completions

To use one of the text models, we need to send a request to the Chat Completions API containing the inputs and our API key, and receive a response containing the model's output.

The API accepts inputs via the `messages` parameter, which is an array of message objects. Each message object has a role, either `system`, `user`, or `assistant`.

- The system message is optional and can be used to set the behaviour of the assistant
- The user messages provide requests or comments for the assistant to respond to
- Assistant messages store previous assistant responses, but can also be written by us to give examples of desired behaviour (however note we can also provide examples within the user message- which is what we tend to do in our workflows)

For example:

```
from openai import OpenAI
client = OpenAI()

response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
```

```
    {"role": "user", "content": "Who won the world series in 2020?"},
    {"role": "assistant", "content": "The Los Angeles Dodgers won the World Series in 2020."
    {"role": "user", "content": "Where was it played?"}
  ]
)
```

Whilst this chat format is designed to work well with multi-turn conversations, in reality we use it for single-turn tasks without a full conversation. So we would normally have something more like:

```
from openai import OpenAI
client = OpenAI(api_key = OPENAI_API_KEY)

completion = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "You are a helpful assistant who specialised in sentime
        {"role": "user", "content": "What is the sentiment of the following text: 'I love rea
    ]
)

print(completion.choices[0].message)
```

The response (defined as `completion` in the code above) of the Chat Completions API looks like the following:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "The sentiment of the text 'I love reading this Handbook' is positive. The
        "role": "assistant"
      },
      "logprobs": null
    }
  ],
  "created": 1677664795,
  "id": "chatcmpl-7QyqpwdfhqwajicIEznoc6Q47XAyW",
  "model": "gpt-4o-mini",
```

```
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 26,
    "prompt_tokens": 13,
    "total_tokens": 39,
    "completion_tokens_details": {
      "reasoning_tokens": 0
    }
  }
}
```

We can see there is a lot of information here, such as the model used and the number of input tokens. You will notice the response is a dictionary and made up of key-value pairs to help organise the relevant information. However, we are mostly focussed on the models output (that is, the assistants reply), which we can extract by running:

```
message = completion.choices[0].message.content
```

19.4 Throughput

An important part of using APIs is understanding the *throughput* - how many requests the API can handle efficiently within a given period.

Broadly, we need to be able to balance cost, model selection, and efficient request handling.

19.4.1 Understanding Tokens and Model Usage

APIs like OpenAI's typically have costs associated with their usage, and this is often measured in tokens. When you input text or data into an API, it is broken down into tokens, which are individual units of language (like parts of words or characters).

- **Input Tokens**: These are the tokens sent to the API (e.g., your prompt to GPT). Every word, punctuation mark, or whitespace counts toward your input tokens.
- **Output Tokens**: These are the tokens returned from the API as a response (e.g., the AI's reply). The longer and more complex the output, the more tokens are consumed.

Managing tokens is crucial because they directly impact the cost of API usage. Different models have different costs per token, with more advanced models being more expensive but often providing better results. For example, as of writing this document (October 2024), the pricing for `gpt-4o-mini` is \$0.150/1M input tokens and \$0.500/1M output tokens compared

to $5.00/1M input tokens and $15.00/1M output tokens for `gpt-4o`. Or in other words, `gpt-4o-mini` is ~30x cheaper than `gpt-4o`! Check out the model pricing information to see the latest costs, and the model pages to see the differences in model capabilities (i.e. context windows, maximum output tokens, training data).

> **Tokens to Words**
>
> - A token is typically about 4 characters of English text.
> - 100 tokens are roughly equivalent to 75 words.

19.4.1.1 How to be more efficient with costs

Despite these costs, there are some strategies we can implement to ensure we make the most of the API usage without unnecessary spending:

- **Remove Duplicate Data**: Ensure your dataset is free from duplicates before sending it to the API. Classifying the same post multiple times is a waste of resources. A simple deduplication process can help reduce unnecessary API calls and cut down on costs, if we then remember to join this filtered dataset (with the model output) back to the original data frame.
- **Clean and Pre-filter Data**: Before sending data to the API, clean it to remove irrelevant or low-value entries. For instance, if you're classifying sentiment on social media posts about mortgages, posts that are clearly not related to the subject matter should be filtered out beforehand. As a rule of thumb it is probably best to run data through the OpenAI API as one of the final analysis steps.
- **Set a Max Token Limit**: Define a `max_tokens` value in your API request to avoid long or unnecessary responses, especially when you only need a concise output (such as a sentiment label or a topic classification). For tasks like classification, where the output is typically short, limiting the tokens ensures the model doesn't generate verbose or off-topic responses, thus reducing token usage.
- **Use the Appropriate Model**: Choose the model that best fits your use case. More advanced models like `gpt-4o` can be expensive, but simpler models like `gpt-4o-mini` may provide adequate performance at a fraction of the cost. Always start with the least expensive model that meets your needs and only scale up if necessary.

Optimise Input Length: Reduce the length of the input prompts where possible. Long prompts increase the number of input tokens and, therefore, the cost. Make your prompts as concise as possible without losing the clarity needed to guide the model effectively.

Batch Processing: Consider grouping multiple similar requests together when appropriate. While asynchronous requests can optimise speed, batching can further reduce overhead by

consolidating similar requests into fewer calls when applicable. Additionally, the OpenAI Batch API can offer cost savings in specific use cases.

19.4.2 Efficient request handling

In addition to costs and model capabilities, there are also rate limits associated with APIs.

These limits are measured in 5 ways that we need to be mindful of:

1. **RPM** - Requests per minute (how many times we call the API per minute). For our current subscription for the vast majority of models we can perform 10,000 RPM.
2. **RPD** - Requests per day (how many times we call the API per day). For our current subscription for the vast majority of models we can perform 14,400,000 RPD (10k x 1440 minutes).
3. **TPM** - Tokens per minute. For our current subscription for the vast majority of models we can perform take in 10,000,000 TPM.
4. **TPD** - Tokens per day. For our current subscription for the vast majority of models we can perform 1,000,000,000 TPD in batch mode.
5. **IPM** - images per minute. For our current subscription `dall-e-2` can take 100 images per minute, and `dall-e-3` can take 15 images per minute.

In reality if it is only a single user calling the API at any one time, it is highly unlikely any of these limits will be reached. However, often there are multiple users calling the API at the same time (working on different projects, workflows etc) and even if we use different API keys, the rate limits are calculated at an organisation level.

There are a couple of ways we can overcome this. The first is to use the (Batch API)[https://platform.openai.com/docs/guides/batch], which enables us to send asynchronous groups of requests. This is actually 50% cheaper than the regular synchronous API, however you are not guaranteed to get the results back (each batch completes with 24 hours). Secondly, we can automatically retry requests with an exponential backoff (performing a short sleep when the rate limit is hit, and then retrying the unsuccessful request). There are a few implementations in Python for this, including the `Tanacity` library, the `backoff` library, or implementing it manually. Examples for these are in the Batch API docs so we will not go into the implementation of them here.

19.4.3 Optimising speed and throughput

In addition to managing rate limits, another critical aspect of API usage is optimising the speed of requests. When handling large datasets or numerous API calls, the time taken for individual requests can add up quickly, especially if each request is handle sequentially. To improve the efficiency of our workflows, we can use **asynchronous calling**.

Asynchronous calling allow multiple requests to be sent concurrently rather than waiting for one to finish before sending the next. This approach is especially useful when processing tasks that are independent of each other, such as classifying individual social media posts.

While asynchronous calling can greatly reduce the time taken to process large datasets, they do not circumvent API rate limits. Rate limits such as Requests Per Minute (RPM) and Tokens Per Minute (TPM) still apply to the total volume of requests, whether sent asynchronous or not. This means that even with asynchronous requests, you need to be mindful of the number of requests and tokens you are sending per minute to avoid hitting rate limits.

19.5 Structured Outputs

As per an update in August 2024, the API introduced a feature called Structured Outputs. This feature ensures the model **will always generate responses that match a defined schema**. While the explanation of *how* they work is beyond the scope of this handbook (there are good resources online from OpenAI), we will discuss *why* they are important and briefly provide a simple example to show how to implement them in a workflow.

19.5.1 Why Not Normal Output?

The output of LLMs is "natural language", which, as text analytics practitioners, we know isn't always in a machine-readable format or schema to be applied to downstream analyses. This can cause us headaches when we want to read the output of an LLM into R or python.

For example, say we wanted to use `gpt-4o-mini`'s out-of-the-box capabilities to identify emotion in posts. We know that a single post can have multiple emotions associated, so this would be a multilabel classification (data point can be assigned to multiple categories, and the labels are not mutually exclusive) problem. We would normally have to include a detailed and complex prompt which explains how we want the response to be formatted, for example `provide each output emotion separated by a semi-colon, such as "joy; anger; surprise"`. Despite this, given enough input data the model will inevitably provide an output that does not follow this format, and provide something like this (one line per input post):

```
joy; anger
joy
This post contains sections of joy and some bits of anger
surprise; sadness
```

We can see the third output here has not followed the prompt instructions. Whilst the other three outputs can be easily read into R using something like `delim = ";"` within the

`read_delim()` function, the incorrect output would cause a lot more issues to parse (or we might even decide to just retry these incorrectly formatted responses, costing more time and money).

Similarly, we might be trying to perform a simple sentiment analysis using a GPT model and ask it to classify posts as either `positive, negative, or neutral`. The output from the API could easily be something like this (one line per input post):

```
neutral
Neutral
positive
Negative
```

Again, we can see a lack of consistency in how the responses are given, despite the prompt showing we wanted the responses to be lowercase.

19.5.2 Benefits of Structured Outputs

So hopefully you can see that the benefits of Structured Outputs include:

1. **Simple prompting** - we don't need to be overly verbose when specifying how the output should be
2. **Deterministic names and types** - we are able to guarantee the name and type of an output (i.e. a number if needed for confidence score, and a classification label that is one of "neutral", "positive", "negative"). There is no need to validate or retry incorrectly formatted responses.
3. **Easy post-processing and integration** - it becomes easier to integrate model responses into further workflows or systems.

19.5.3 Simple Example Implementation

To showcase an example, let's say we want to classify posts into emotions (joy, anger, sadness, surprise, and fear) in a multilabel setting, to ensure the response is consistently formatted. If you're familiar with coding in python you might recognise `Pydantic`, which is a widely used data validation library for Python.

1. Define the schema

We define a schema that includes only the emotion labels. This schema ensures that the model returns a list of emotions it detects from the text, following the structure we define. We do this by creating a Pydantic model that we call `EmotionClassification` which has one field, `emotions`. This field is a list that accepts only predefined literal values, allowing multiple emotions to be included in the list when detected.

```
from pydantic import BaseModel
from openai import OpenAI
from typing import Literal

class EmotionClassification(BaseModel):
    emotions: list[Literal["joy", "anger", "sadness", "surprise", "fear"]]  # List of detecte
```

2. Call the model

We then call the model as before, but importantly we include our schema via the `response_format` parameter.

> **Note**
>
> Here we use `client.beta.chat.completions.parse` rather than `client.chat.completions.create` because Structured Output is only available using the `.beta` class.

```
# Sample text to classify
text_to_classify = "I was so happy when I got the job, but later I felt nervous about startin

client = OpenAI(api_key = OPENAI_API_KEY)

completion = client.beta.chat.completions.parse(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "You are an emotion classifier. For the provided text,
        {"role": "user", "content": text_to_classify}
    ],
    response_format=EmotionClassification,
)

emotion_results = completion.choices[0].message.parsed
```

If we view the output of this we see we get a nice output in JSON format:

```
EmotionClassification(emotions=['joy', 'fear'])
```

19.6 The Playground

The Playground is a tool from OpenAI that you can use to learn how to construct prompts, so that you can get comfortable using querying the model and learning how the API works. **Note however that using the Playground incurs costs. While it's unlikely you will rack up a large bill be analysing large corpora within the playground, it is still important to be mindful of usage**.

20 Resources

20.1 Data Visualisation

- Capture Cookbook
- R Graph Gallery

20.2 Literate Programming

- Quarto Guide
- Quarto Markdown Basics
- RStudio R Markdown guide
- Visual R Markdown

20.3 Package Development

- R Packages Book
- pkgdown
- roxygen2
- usethis
- devtools
- testthat
- Tidyverse Design Guide
- Happy Git with R

20.4 YouTube Channels

These channels will help you see the power and flexibility of data analysis & science in R:

- Julia Silge
- David Robinson

20.5 Shiny

- Mastering Shiny
- Engineering Enterprise-grade Shiny Applications with Golem
- Outstanding Shiny Interfaces
- Modularising Shiny Apps, YT
- Presenting Golem, YT
- Styling Shiny, Joe Chen, YT

20.6 Text Analytics

- Text Mining with R (Tidytext)
- Supervised Machine Learning for Text Analysis in R
- Speech & Language Processing
- Natural Language Processing with Transformers - we have a physical copy floating around the office