Data Science at SHARE Creative

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 SHARE Creative. 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.

The team is made up of:

Mike Tapp: Data Director

Jack Penzer: Global Data Product Lead

Jamie Hudson: Senior Data Scientist

Aoife Ryan: Data Scientist

Tim Mooney: Jr. 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.

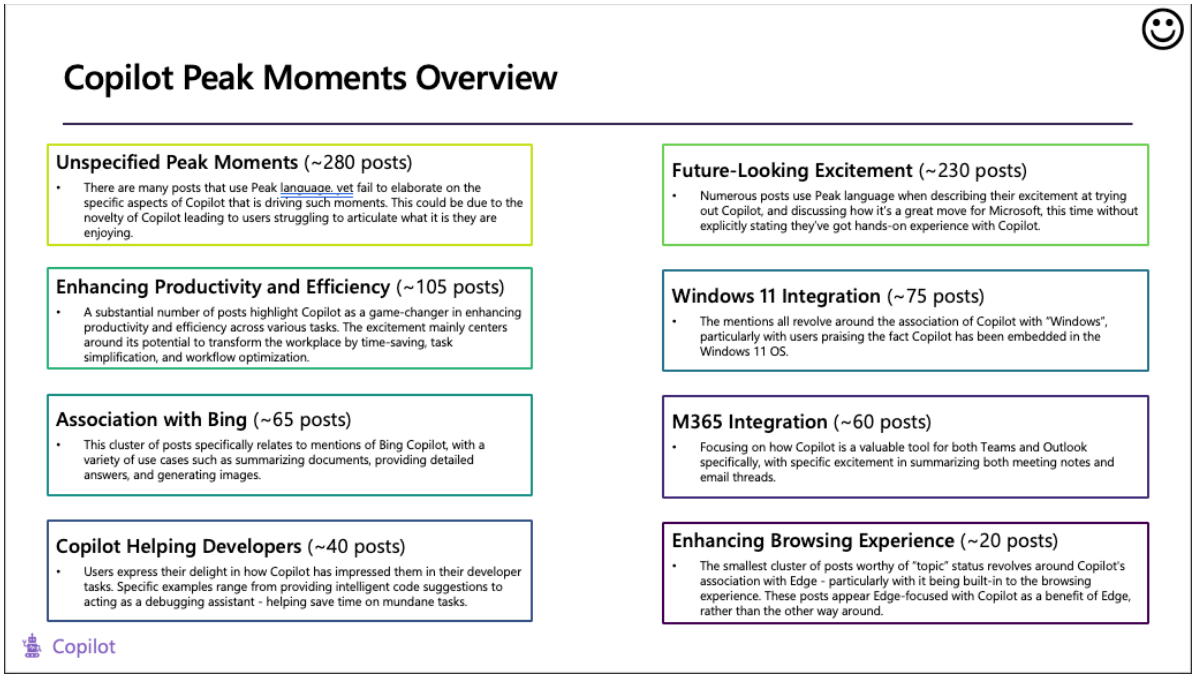


Figure 3.1: Screenshot from a Peaks and Pits project showcasing the identified Peak moments for a product at a high level

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

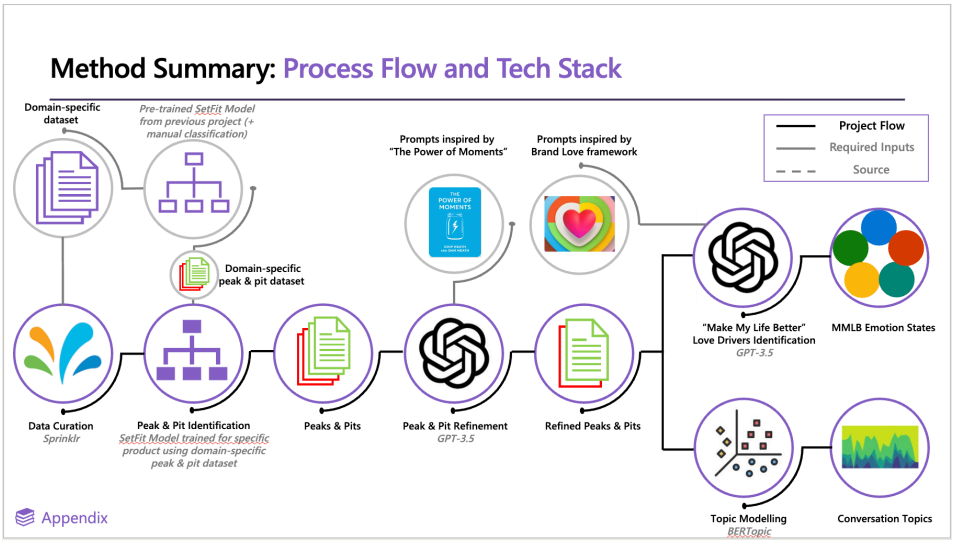


Figure 3.2: Schematic workflow from Project 706 - Peaks and Pits in M365 Apps

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:

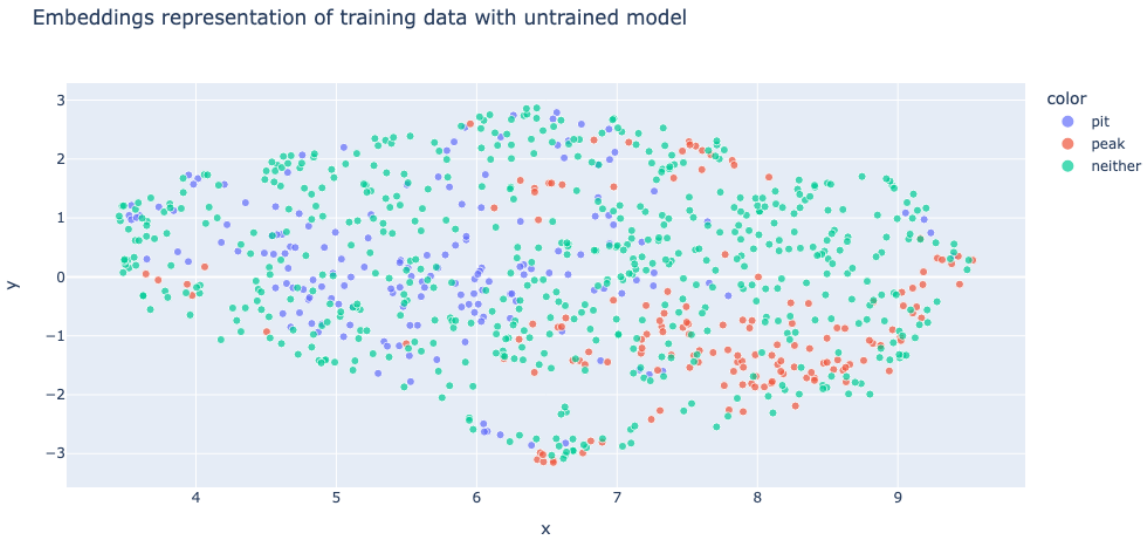


Figure 3.3: Un-finetuned embedding model

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 visualization. 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 colour 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).

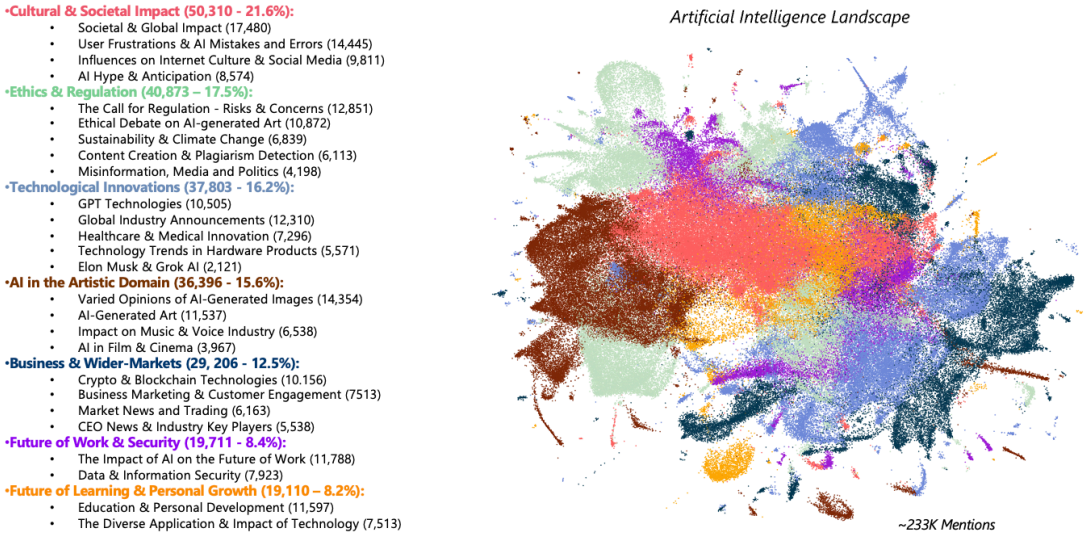


Figure 4.1: Screenshot Taken from the Final Output of an AI Landscape Microsoft Project - Q2 FY24

4.3 How to get there

As promised, we will provide some more context surrounding the required steps taken, so that a reader may replicate and implement the methods mentioned throughout, 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, 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 Modeling/Clustering: Scientifically modeling 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 visualized 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.

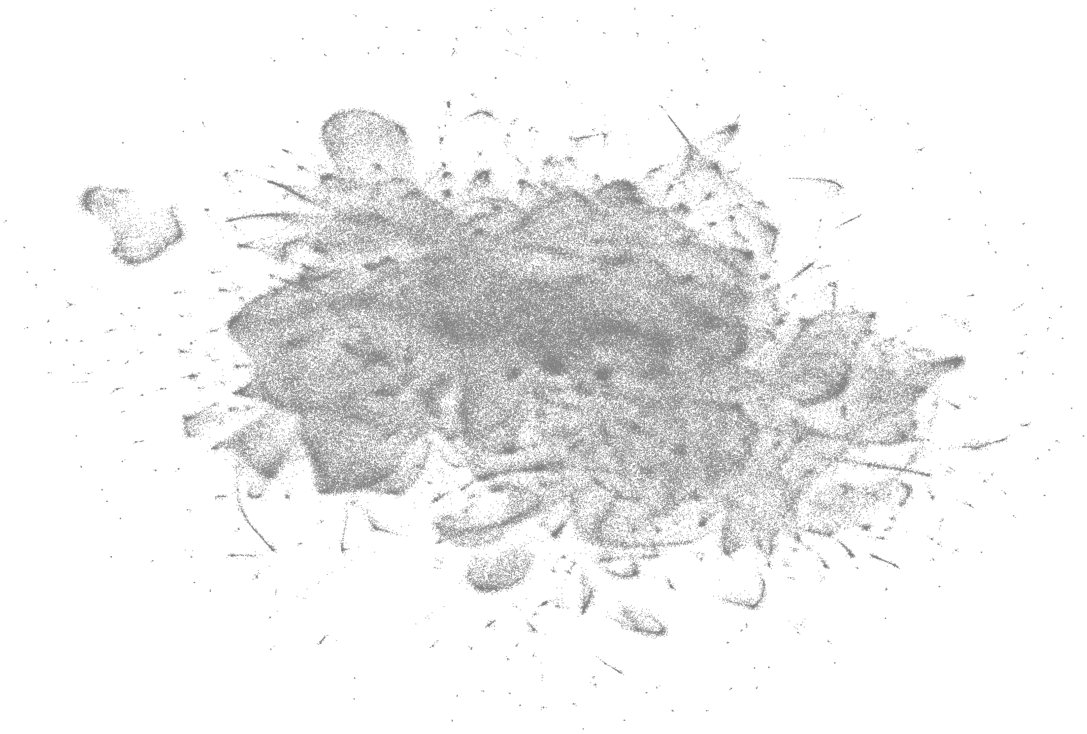


Figure 4.2: Grey Colourless Landscape Plot from an AI Landscape Microsoft Project - Q2 FY24

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 K-Means and HDBSCAN. However, the example we have shown uses K-Means, 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.

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, to 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-fit models, it is essential to do so with the exact same versions of important libraries and Python. The reason being 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 K-Means CLustering

`n_clusters`: K-Means 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 mentioned 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.

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.

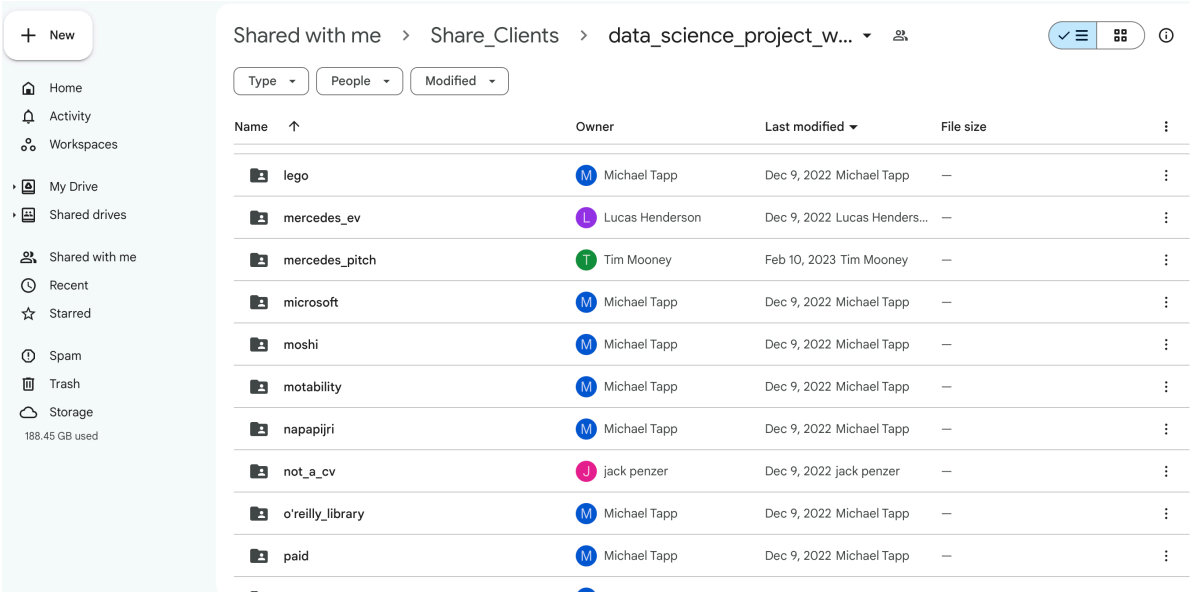


Figure 5.1: Screenshot of the `data_science_project_work/` directory with client-specific sub-directories

> **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/`).

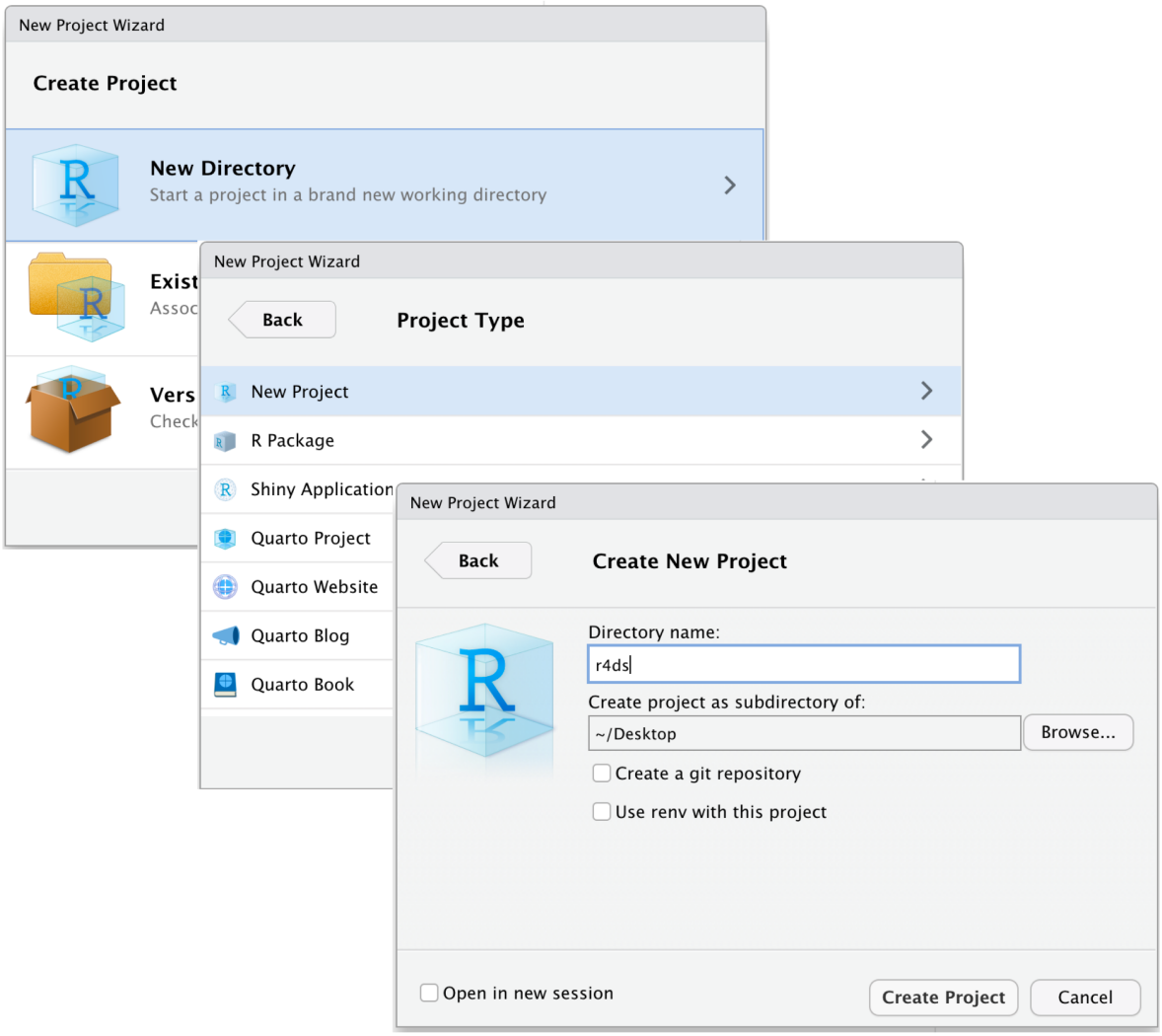


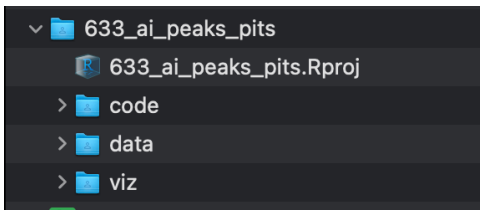
Figure 5.2: Steps to create a new project, taken from R for Data Science (2e) https://r4ds.hadley.nz/workflow-scripts.html#projects

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`).

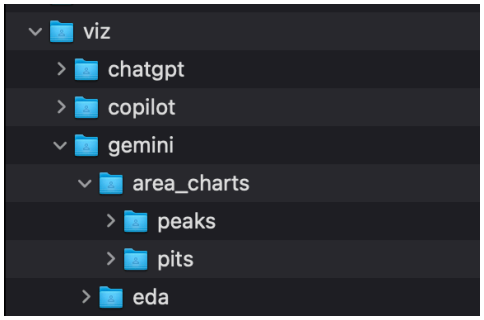


Figure 5.3: Example viz directory hierarchy for a Peaks and Pits project

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

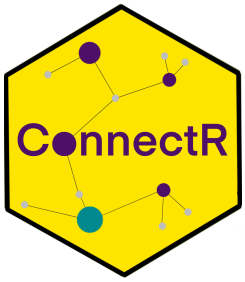
7 Our Packages

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 SHARE 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 SHARE 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 SHARE'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.

Part V

Tips and Tricks

8 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.

8.1 Why are we here?

At SHARE & Capture, 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.

8.2 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
```

8.2.1 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?
- [ ]
- [ ]

8.3 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.

8.4 On managing complexity

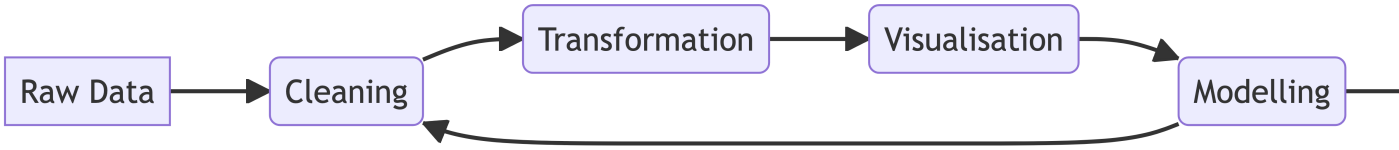
> "…let's think of code complexity as how difficult code is to reason about and work with."

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.

8.5 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.

8.5.1 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.

8.5.2 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.

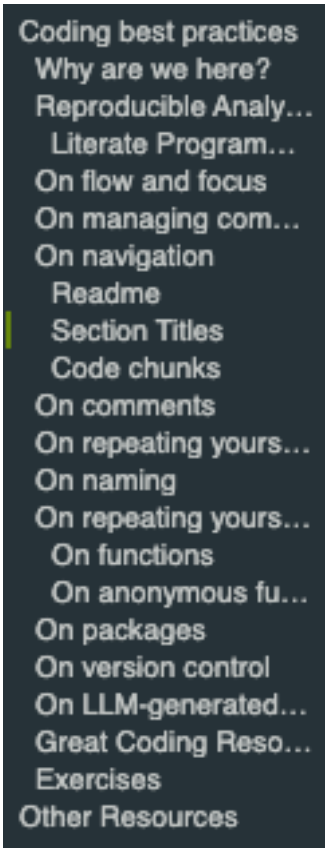


Figure 8.1: Rstudio Outline

8.5.3 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.

8.6 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).

8.7 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

8.8 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.

8.9 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.

8.9.1 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.

8.9.2 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.

8.10 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 SHARE package.

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

see also: Package Resources

8.11 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.

8.12 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.

8.13 Great Coding Resources

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

8.14 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!

9 Other Resources

cognitive strain coding is thinking not typing Google Code Health

10 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

10.1 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*

10.1.1 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.

10.1.1.1 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.

10.1.1.2 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")
```

10.1.1.3 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)
```

10.1.2 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)
```

10.2 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:

10.2.1 Remove punctuation

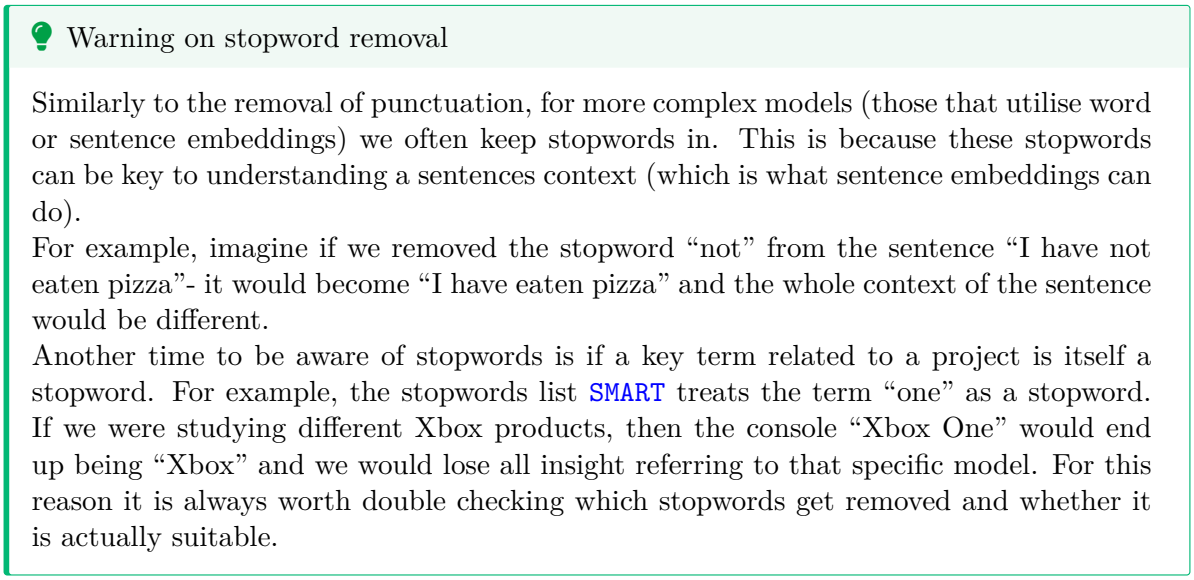
Often times we will want punctuation to be removed before performing an analysis because they *tend* to not be useful for text analysis. This is particularly the case with more 'traditional' text analytics, where an algorithm will assign punctuation marks a unique numeric identify just like a word. By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

10.2.2 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.



10.2.3 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.

10.2.4 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.

10.2.5 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.

10.2.6 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.

10.2.7 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.

10.3 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.

11 Resources

11.1 Literate Programming

- Quarto Guide
- Quarto Markdown Basics

11.2 Package Development

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

11.3 Reproducible Research

- RStudio R Markdown guide
- Visual R Markdown

11.4 YouTube Channels

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

- Julia Silge
- David Robinson

11.5 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

11.6 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