Technical Guide

MyTrip - A Trip Made Just For You

Student 1: James Fallon - **Student Number**: 18394123
Student 2: Alex O Neill - **Student Number**: 18414882

Project Supervisor: Gareth Jones

Date Completed: 24/04/2022

CA400 - Technical Guide

Table of Contents

1. Abstract

MyTrip is a web application for travel enthusiasts. It acts as a recommender system for people going on trips to different cities around the world that want to experience new restaurants and activities that suit their interests. The goal of our application is to improve the travelling experience, recommending people restaurants and activities that they may not have been to before. MyTrip uses machine learning algorithms such as content based filtering, collaborative filtering and sentiment analysis to generate recommended attraction based on the user's interests.

2. Motivation

The motivation for this project came from using trip planning websites that already exist but we found that they appeal to a very particular group of people. Most of these websites cater towards family attractions and a lot of the time they are attractions most people are already aware of. We wanted to try solve this problem by recommending users attractions, restaurants and activities based on their interests.

We were also very interested in learning more about machine learning and how we could incorporate it into a meaningful app that we could use in our lives

3. Research

Due to the fact that we had never worked with machine learning before, we wanted to understand the fundamentals of machine learning first. This area of computing requires a large amount of research before beginning an implementation. Good research leads to good planning and this was out aim as we started.

We began with a plural-sight course on the basics of ML and an introduction to tensor-flow. This course thought us the basics of machine learning.

Basics of ML

- Corpus (collection of data) → ML based Classifier → Classification
- The classification keep getting fed back into the classifier (optimisation)
- This is called the feedback, loss/objective function (improve weights and biases)
    - this is the training of the neural network
- This feedback loop is how we improve the functions parameters to tune better predictions
- The feedback loop can also tell us what features are important for classification

Neural Networks

- A neural network is a function that is used to find the most important features within a collection of input data
- A NN is a collection of Neurons.
- Neuron → building blocks that learn from data, a maths function

Layers

- Input → hidden → output
- The hidden layers are layers in the network that alter the data, but we do not act on the layer directly
- Series of inputs (Xn) → Math func → (Yn)
- Input changes output
- Edges have weights → the weight is the strength of connection between neurons
- The higher the sensitivity of second neuron to the output of the first neuron, the higher the value of the weight

Computational Graph

- This refers to what we know as the visual representation of a neural network. An input layer, output layer with hidden layers contained in between.
- Tensors → Edges of data (connections between neurons)
- edges in the graph are the data that is able to be operated on
- input → math func.(neuron) → output

Model parameters

- This is the actual “training” of the model
- Finding the models parameters
- By training a model we are adding weights to all edges
- Trained model → all edges have weights

After using tensorflow in this course, we were introduced to keras as a more user-friendly version of tensor-flow and was a great to tool to begin learning machine

learning.

tf.Keras

- High level API which helps create NN using models and layers
- Keras API is part of tensor-flow
- models, building NN, dynamic execution graphs
- Storing and retrieving models on disk

Dynamic and Static Computational Graphs

Everything in tensor-flow is a graph, and the graphs are made up of tensors (nodes). These nodes are functions and control data. These tensors operate on and manipulate data. There are two types of development when using tensor-flow, static and dynamic. Static is best for deployment while dynamic is best of development.

1. **Static**
    a. Define and then execute
    b. Compiled before execution
    c. This is like Java and C
    d. First define computation and then run
    e. Computations first defined using placeholders
    f. Explicit compile step where the graph is created
    g. Harder to debug
    h. Not good for experimentation
    i. Much easier to optimise static graphs
2. **Dynamic**
    a. Execution and operation are simultaneous
    b. No compilations
    c. This is like python, runtime
    d. Computations run as they are defined

e. Performed on real operands
f. PyTorch is dynamic
g. Graph is already in executable format
h. Writing and debugging is easier
i. More flexible → Easier to experiment
j. Immediate results visible to users
k. These are much harder to optimise

What is the basics of the training process?

- We do this by calculating gradients
- Gradient tape library from tensor-flow
- Manually train a model using these gradients

Training:

- The weights and biases is what we are looking to find during the training exercise
- We are looking for the regression line → this is the best fit regression line → y = mx + c
- This line can be used to predict y values given x values to help is predict the most probable solution for a problem
- Minimise the mean square error (MSE)
  - This is the squares of the distances of the points from the regression line
- Gradient decent optimisation
- Weight and biased values associated with the neuron
- We will get a 3D graph
  - Best regression model → where MSE has the smallest value
  - Use the regression model to find the best values from b and weight.

    - We can find this using gradient decent → converge on the best values for our parameters
    - When we say “training a model”, we refer to finding the best values for our parameters using the gradient decent optimisation algorithm

Forward pass

- Giving a model data and receive a prediction, using the current models parameters
- Compare actual values from training data with the prediction
- Use these to calculate the error/loss function of the model
- Error is then fed into the optimiser
- Tweaks the model parameters to minimise the error
- This is done by calculating the gradient (with the optimiser)

Backward pass

- A backward pass is used to optimise the parameters after a forward pass
- We start at the Last layer and update it first
- After a backward pass, we do a forward pass again
- The backward pass allows the weights and biases of the biases to converge on their final values
- this is the training operation of the neural network

Calculating Gradients and Using Gradient Tape

- We use the MSE to check our values that we generate in our training mentioned about against the real values.
- Different between Actual values from your training data and the predicted values that are outputted
- We want to minimise the loss
- Loss = Predicted y - actual y (from training data)
- Gradient → list (vector of partial derivatives)

- Partial derivatives of loss with regard to parameter W → How sensitive is the loss to changes in W?
- While all other input and parameters are constant → how much does the loss value change when we change W?

Keras usage

At this point in our research we were getting overwhelmed by the amount we had to learn and more importantly how we were going to adapt it to our project implementation. Once we began to use keras, the implementation became a lot simpler as it did most of the heavy work and kept our code tidy and easy to understand and modify.

Keras building blocks

- Layers (Collection of Neurons)
- Models (Consist of Layers)
- Sequential models
- Functional API's → more complex models
- Model subclassing → granual control over classes
- We can build our own Custom layers

Sequential Models in Keras

These are simple stacks of layers. They are mainly used for simple models. They are build into keras and are great for building simple stacks of layers. (tf.keras.Sequential)

We found the simple make up of a keras model to be perfect for out implementation:

1. Instantiate model (Container for layers)
2. Specify shape of first layer (Equal to shape of input)
3. Add layers (Several standard steps)
4. Compile Model (optimisation, loss function, model.compile())
5. Train model (epochs, batch size, training data, model.fit())
6. Use model to predict with test data (model.predict())

This course then went more in depth of Functional models in keras and their subclassing counterpart, but we decided these were out of the scope of our project, as by this part in our research we wanted to start implementing what we had learned so far.

3.1 Building Recommender Systems with Machine Learning

We then moved onto the application of our knowledge through a uDemy course on building recommender system with machine learning. We went in as if we knew nothing at all, and tried to lay a great foundation that our implementation could be build upon.

What is a recommender system?

> A recommender system does NOT recommend arbritary values. This can be used to describe machine learning in general. It is not a general purpose algorithm. Recommender systems are very specialised. They provide a recommendation based on their behaviour and the behaviour of other people, or similar properties to the things you like.

Types of recommenders

1. Past purchasing decisions influence new recommendations
2. Actions predict purchases
3. Historical patterns
4. Recommending things, content, music, people, search results

How do they work?

1. They try to understand people
2. Merge user data with everyone elses
3. Explicit feedback → Rating content (likes, dislikes)
    a. Requires extra work for user to give us data in their own time
    b. Different standards can lead to biases
    c. Cultural differences can sway the data
4. Implicit behaviour
    a. Passive data collection
    b. Things you click on (clickbait problem)

c. Thing you purchase (reliable)

d. Things you consume

e. Need good data to work with

Top-N recommenders

- Looking for things they are likely to love, based on what has the highest probability of getting a good rating from a user

3.2 Basics and the Cosine Similarity Metric

- The cosine Similarity measure
- Each ‘property’ has a score (0.0 - 1.0)
- Plot the scores on a graph (comparing two genres)
- Get angle between two movies
- 0 means not at all similar
- 1 means they are the same thing
- cosine of angle is the similarity score
- cosine of 0 degrees is 1.0, so the attractions are identical in our implementation
- cosine of 90 degrees is 0.0, so the attractions have nothing in common
- Euclidean distance → distance between two points as we plot them
- Pearson correlation is another metric we decided not to use
- Scaling up to multiple dimensions
  - 18 dimensional points have 18 points
  - we decided to settle on four to six dimensional points as we saw diminishing returns which will be discussed in our testing below.
- Computing the cosines between vectors in multidimensional space is how we implemented out content based recommender, which can be seen in our code samples below.

3.3 K-Nearest Neighbours and Content Recommendations

- Basing similarity on the properties of the attraction
- Always start with a baseline
- Absolute values can be useful

3.4 How do we turn this into actual ratings?

- k-nearest neighbours
- similarity scores for rated movies to this movie
- nearest → highest content based similarity scored to the movie were making a prediction for (highest cosine similarity)
- sort → 10 attractions whose properties most closely match the item the user has told us they enjoy, we are trying to evaluate for this user
- predicting based on the properties of their neighbours

3.5 A note on Implicit Ratings

- Implicit data is plentiful, as a user generates it without knowing
- More powerful then explicit based ratings (such as purchases)
- Ensure to use consistent values (comparable scores)
- A user not giving a rating, does not equal a negative rating → it's just missing data
- Purchases are good, clicks not so much (not a reliable source)
- No money on the line → how high quality is the rating?

We decided not to use implicit as the explicit implementation gave us string results when tested.

3.6 Collaborative Based Filtering

Once we were happy with our knowledge for content based filtering, we then decided to take a look at collaborative based filtering as well as a way to diversify our recommendations. Since we a requirement of our project is that a user would be able to rate items, we knew that collaborative based filtering would be a great fit if we could implement it appropriately.

1. User interests
2. Candidate generation based on similar ratings
3. Candiate ranking
4. Filtering

As we began to shape our dataset we came across some issues that we discuss below in our problems solved section.

3.7 Collaborative filtering basics

- Embeddings
- CosSin(x,y) (user based Pearson similarity)
    - Works well to measure the similarity of users
    - Does not work well with sparse data (new users)
- Pearson similarity (Item based)
    - Measuring the similarity between people based on how much they diverge from an average persons behaviour
- Mean squared difference (MSD) → calculate similarity of two users or items
    - CosSim still works better in practice
- Collaborate filtering types
    1. User Based
    2. Item based
- Jaccard Similarity
    - Size of intersection divided by the unions of both users
    - Used for implicit ratings (clicks and purchases)

3.8 Item Based Collaborative filtering

- Items as rows and users as columns
- CosSim between item vectors for users

- Item based is what Amazon uses
- move to an AB test
- Using a rating threshold (above 4 stars) is seemingly more accurate then not using it
- Hit rate is low for item based CF evaluation
- **Recommending obscure items only leads to a distrust from the user**
- Focusing on accuracy alone is not a good idea

We decided to focus on user based collaborative filtering as we were more focused on the user rather than the items. You would use item based for a website that recommends you items based on other times you already like such as amazon. This did not fit our needs.

3.9 Matrix Factorisation

We then came across matrix factorisation.

Matrix factorisation is when we divide a large matrix into tow smaller ones, that are easier to store and compute values from.

Why?

- Collaborative filtering is sensitive to sparse data and noisy data
- CF needs a lot of data that is nice and clean
- Start by working backwards
- Filling in a really sparse matrix 2d
- Users as rows, attractions as columns
    - we realised that this matrix was too large to realistically worked with and we used solutions discussed later in this technical specification

3.10 Deep Learning

We then began to undertake more research in deep learning to see how it may improve our project. What we realised is we had been doing deep learning all along but the inner workings were hidden by the libraries we were using. In the context of matrix factorisation to gain predictions, we use embeddings to find candidate predictions.

We went back and implemented an early model without using keras. This model had the weights exposed and the biases calculated by hand. We gained an understanding for the workings of the models we would have to implement but the code became unnecessarily complicated and we decided to use keras and its many libraries for the rest of our research.

We began to work on the following models:

1. multi-class classification
2. binary classification
3. training a CNN
4. training a RNN
5. training a GAN

At this stage, we began to gain an understanding for some of the parameters involved in tuning a model, alongside activation functions and their purpose:

1. Learning rate
    a. Gradient decent based
    b. Start at some random point, sample some different solutions (weights), seeking to minimise some cost function
    c. Distance between samples is the learning rate
    d. Too high? → overshoot optimal solution
    e. Too small? → Too time intensive
    f. Learning rate is a hyper parameter

2. Batch size
    a. Amount of training samples used in each epoch
    b. Small sizes → work out of ‘local minima’
    c. Too large → can get stuck in wrong solution at random

Types of activation functions:

1. reLU
2. Softmax
3. Sigmoid

We then came across some regularisation techniques to prevent overfitting to our training set:

1. Too many Layers/Neurons?
    a. Sometimes fewer neurons or layers may be used to prevent overfitting
    b. Aim for same accuracy without overfitting
2. Dropout
    a. Removing neurons to make the model spread out its learning
3. Early stopping
    a. Once the validation accuracy has levelled out, we stop

3.11 Sentiment analysis

Introduction to Natural Language Processing

The first task of the sentiment analysis research was learning about natural language processing. We learned this information through a Udemy course which helped us understand how we would implement our sentiment analysis machine learning algorithm.

Some of the key points we learned were:

- How to program computers to process and analyze large amounts of natural language data
- Computers are very good at handling direct numerical information. But what about text data?

- A computer needs specialized processing techniques in order to “understand” raw text data
- Text data is highly unstructured and can be in multiple languages
- Natural language processing attempts to use a variety of techniques in order to create structure out of text data
- These techniques are built into libraries such a Spacy and NLTK

Example use cases:

- Sentiment analysis of text reviews - if positive or negative

3.12 Machine learning Introduction

This was our first project in machine learning and so the fundamentals of machine learning had to be learned by both of us.

Machine learning overview:

Machine learning is a method of data analysis that automates analytical model building.

Using algorithms that iteratively learn from data, machine learning allows computers to find hidden insights without being explicitly programmed where to look.

We learned about supervised machine learning which are trained using labeled examples, such as an input where the desired output is known. For example, a piece of text such as a review which would be labelled as Positive vs Negative. The learning algorithm receives a set of inputs along with the corresponding correct outputs, and the algorithm learns by comparing its actual output with the correct outputs to find errors. It then modifies the model accordingly. Supervised learning is commonly used in applications where historical data predicts likely future events and this was exactly what we needed for our sentiment analysis model.

3.13 Classification Metrics

Having data labelled and it’s features is important as supervised machine learning uses this information to try to classify what the label will be for an unknown feature, for example Positive or Negative sentiments of review texts.

In classification, the model can only achieve two results:

- Correct in its prediction

- Incorrect in its prediction

3.14 Model Evaluation:

We learned how we would evaluate our sentiment analysis model through the use of the following metrics:

Accuracy:

Is the number of correct predictions made by the model divided by the total number of predictions.

If X test set was 100 reviews and our model correctly predicted 80 reviews, classifying as positive or negative correctly, then we have 80/100 = 80% accuracy.

Recall:

This is the ability of a model to find all the relevant cases within a dataset. The number of true positives divided by the number of false negatives.

Precision:

Identifying only the relevant data points. True positives divided by the number of true positives plus the number of false positives.

F1 score:

Mean of precision and recall

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

Confusion matrix:

This contains two main categories:

- True condition
- Predicted condition

Correctly classified to class 1: True Positive

Correctly classified to class 2: True Negative

Incorrectly classified to class 3: False Positive

Incorrectly classified to class 2: False Negative

These metrics allowed us to compare the predicted values with the true values and showed how accurate our model was.

3.15 Sentiment Analysis Using VADER

We initially explored text classification and using it to predict labels on pre-labeled movie reviews. But what if we don't have those labels? We wanted to learn more about unsupervised learning and with some research we found VADER (Valence Aware Dictionary for sEntiment Reasoning) is a model used for text sentiment analysis that is sensitive to both polarity (positive/negative) and intensity (strength of emotion).

It is available in the NLTK package and can be applied directly to unlabeled text data. VADER sentiment analysis relies on a dictionary which maps lexical features to emotion intensifies called sentiment scores.

The sentiment score of a text can be obtained by summing up the intensity of each word in the text. For example, words like "love", "like", "enjoy" and "happy" all convey positive sentiment. VADER is intelligent enough to understand basic context of these words, such as "did not love" as a negative sentiment. It also understands capitalization and punctuation, such as "LOVE!!!"

After carrying out a tests on movie reviews using VADAR, we came to the conclusion that it couldn't judge the movie reviews very accurately. It demonstrated one of the biggest challenges in sentiment analysis i.e. Understanding human semantics. Many of the reviews had positive things to say about a movie reserving any final judgement to the last sentence. Unfortunately these reviews were therefore often predicted as a negative sentiment. We decided to continue researching to find a better solution.

3.16 Topic-Modelling:

We next looked into topic-modelling. Topic modelling allows for us to efficiently analyze large volumes of text by clustering documents into topics. A large amount of text data is unlabeled meaning we won't be able to apply our previous supervised learning approaches to create machine learning models for the data. Topic modeling puts it on us to try to discover those labels through modeling. If we have unlabeled data, then we can attempt to "discover" labels. In the case of text data, this means attempting to discover clusters of documents, grouped together by topic. It's important to remember that we don't know the correct topic. All we know is that the documents clustered together share similar topic ideas. It is up to the user to identify what these topics represent.

Latent Dirichlet Allocation

2 assumptions:

- Documents with similar topics use similar groups of words
- Latent topics can then be found by searching for groups of words that frequently occur together in documents across the corpus.

Documents are probability distributions over latent topics.

- any particular document is going to have a probability over a given amount of latent topics

e.g. Doc1 has the highest probability of belonging to topic 2 etc. Not saying definitively that Doc1 belongs to a specific topic, instead we are modeling them over the probability distributions.

Latent Dirichlet Allocation represents documents as mixtures of topics that spit out words with certain probabilities. It assumes that documents are produced in the following fashion:

- Decide on the number of words N the document will have.
- Choose a topic mixture for the document (according to a Dirichlet distribution over a fixed set of K topics). e.g. 60% business, 20% politics, 10% food.
- Generate each word in the document by:
    - First picking a topic according to the multinomial distribution that we sampled previously (60% business, 20% politics, 10% food)

- For example, if we selected the food topic, we might generate the word “apple” with 60% probability, “home” with 30% probability and so on.

Assuming this generative model for a collection of documents, LDA then tries to backtrack from the documents to fund a set of topics that are likely to have generated the collection.

Now imagine we have a set of documents.

We’ve chosen some fixed number of K topics to discover, and want to use LDA to learn the topic representation of each document and the words associated to each topic.

Go through each document, and randomly assign each word in the document to one of the K topics.

This random assignment already gives us both topic representations of all the documents and word distributions of all the topics (note, these initial random topics won’t make sense).

Now we iterate over every word in every document to improve these topics.

p(topic t | document d) = The proportion of words in document d that are currently assigned to topic t

Reassign w a new topic, where we choose topic t with probability p(topic t | document d) * p(word w | topic t)

This is essentially the probability that topic t generated word w.

After repeating the previous step a large number of times, we eventually reach a roughly steady state where the assignments are acceptable.

At the end we have each document assigned to a topic.

We also can search for the words that have the highest probability of being assigned to a topic.

We end up with an output such as:

Document assigned to Topic #4. Most common words (highest probability) for Topic #4:

- ['cat', 'vet', 'birds', 'dog']

It is up to the user to interpret these topics.

3.17 Deep Learning

After learning about two possible routes we could take (supervised or unsupervised learning), the next step was to try understand deep learning using neural networks for sentiment analysis.

Deep learning is the process of using artificial neural networks to learn tasks using networks of multiple layers. It is inspired by the structure of the human brain, consisting of a large number of information processing units, called neurons, organized into layers which work together. It can learn to perform tasks such as classification by adjusting the connection weights between neurons which mimic the human brain.

Weights represent the strength of the connection between neurons. If the weight from node 1 and node 2 has greater magnitude, it means that neuron 1 has greater influence over neuron 2. A weight brings the value of the input down. Weights near zero means changing this input will not change the output. Negative weights mean increasing this input will decrease the output. A weight decides how much influence the input will have on the output. It is an extra input to neurons and it is always 1, and has it's own connection weight. This makes sure that even when all the inputs are none (all 0's) there's gonna be an activation in the neuron.

We have to cater for if the inputs are zero. If we multiple anything by zero it will always equal zero. Therefore a bias is introduced. These are then passed into an activation function

Forward Propagation—Forward propagation is a process of feeding input values to the neural network and getting an output which we call predicted value. Sometimes we refer forward propagation as inference. When we feed the input values to the neural network's first layer, it goes without any operations. Second layer takes values from first layer and applies multiplication, addition and activation operations and passes this value

to the next layer. Same process repeats for subsequent layers and finally we get an output value from the last layer.

Back-Propagation—After forward propagation we get an output value which is the predicted value. To calculate the error (loss) we compare the predicted value with the actual output value. We use a loss function to calculate the error value.

Then we calculate the derivative of the error value with respect to each and every weight in the neural network.

In chain rule first we calculate the derivatives of the error value with respect to the weight values of the last layer. We call these derivatives gradients and use these gradient values to calculate the gradients of the second last layer. We repeat this process until we get gradients for each and every weight in our neural network. Then we subtract this gradient value from the weight value to reduce the error value.

3.18 One hot encoding

One hot encoding can be defined as the essential process of converting the categorical data variables to be provided to machine and deep learning algorithms which in turn improve predictions as well as classification accuracy of a model. One Hot Encoding is a common way of preprocessing categorical features for machine learning models.

This type of encoding creates a new binary feature for each possible category and assigns a value of 1 to the feature of each sample that corresponds to its original category.

For example, we can encode our “positive” and “negative” labelled sentiments into 1 and 0 respectively.

3.19 What is overfitting

Overfitting refers to the model that models the training data too well and begins to memorize the data rather than learn from it. This is a common pitfall for many in machine learning and one in which we had to learn to try and overcome. The model can often memorize the data patterns, noise and random fluctuations.

3.20 Activation Functions

Softmax:

Softmax is **a mathematical function that converts a vector of numbers into a vector of probabilities**

3.21 Adam Optimizer:

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

- Straightforward to implement.
- Computationally efficient.
- Little memory requirements.
- Invariant to diagonal rescale of the gradients.
- Well suited for problems that are large in terms of data and/or parameters.
- Appropriate for non-stationary objectives.
- Appropriate for problems with very noisy/or sparse gradients.
- Hyper-parameters have intuitive interpretation and typically require little tuning.

An optimizer is, it's simply the mechanism that constantly computes the gradient of the loss and defines how to move against the loss function in order to find its global minima and therefore, find the best network's parameters (the model kernel and its bias' weights).

3.22 Introduction to Keras and Tensorflow for Deep Learning

After getting an understanding of the theory of deep learning, it was important to try and apply the knowledge in practice.

We used the Iris Dataset that is built in to Tensorflow. This dataset contains three classes of fifty instances each, where where each class refers to a type of iris plant. It contains measurements of the three classes:

- sepal length
- sepal width
- petal length
- petal width

Reading in the Dataset

```
from sklearn.datasets import load_iris

iris = load_iris()

X = iris.data
print(X)
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],

#Get the features - in this case the flower measurements
#numpy array with the 4 measurements lined up:
#sepal length in cm
#- sepal width in cm
#- petal length in cm
#- petal width in cm
```

```
y = iris.target
print(y)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
#The class of plant sorted into classes 0, 1, 2
```

Because of the way Keras and neural networks work in general we're actually going to want to convert this to a categorical labeling system essentially one hot encoding. This means, we want a vector that's going to be zero for everywhere that this class doesn't match up. i.e., Class 0 is going to be one, index zero

class 0 -> [1,0,0]
class 1 -> [0,1,0]
class 2 -> [0,0,1]
This is known as one hot encoding

```
from tensorflow.keras.utils import to_categorical

y = to_categorical(y)
print(y)
```

```
array([[1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
         [1., 0., 0.],
```

```
          [1., 0., 0.],
          [1., 0., 0.],
          [1., 0., 0.],
          [1., 0., 0.],
```

Split the Data into Training and Test datasets

We need to now split all our data into a training and a test set. This is done so that we have enough data to train our model but also enough data to test the model on unseen data. Train test split will also shuffle the order which is good as we don’t want the model to train on ordered data.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

Standardizing the Data

Usually when using Neural Networks, you will get better performance when you standardize the data. Standardization just means normalizing the values to all fit between a certain range, like 0-1, or -1 to 1. It is important to do this as it will help the weights and biases from growing too large.

```
from sklearn.preprocessing import MinMaxScaler

scaler_object = MinMaxScaler()

scaler_object.fit(X_train)
#only fit it to the train data as we don't want to assume prior knowledge of the test data

scaled_X_train = scaler_object.transform(X_train)

scaled_X_test = scaler_object.transform(X_test)
#Now we have the data scaled
```

Building the Network with Keras

Now it was time to build the network with Keras

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential() #create the model
model.add(Dense(8, input_dim=4, activation='relu'))

#add layers to the model.
#8 neurons (multiple of features)
#4 input dimension (features (categories))
#activation function = relu
model.add(Dense(8, input_dim=4, activation='relu'))
#output layer
model.add(Dense(3,activation='softmax'))

#at 3 neurons, each neuron is going to have a probability of belonging to a certain class [0
#This is what softmax does - converts a vector of numbers into a vector of probabilities

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
#compile the model.

#We choose a loss, this depends on what you are performing. Here we are performing categoric
#Adam is an optimization algorithm that can be used instead of the classical stochastic grad
```

Fit (Train) the Model

```
model.fit(scaled_X_train, y_train, epochs=150, verbose=2)
```

```
Epoch 1/150
 - 0s - loss: 1.0926 - acc: 0.3400
Epoch 2/150
 - 0s - loss: 1.0871 - acc: 0.3400
Epoch 3/150
 - 0s - loss: 1.0814 - acc: 0.3400
Epoch 4/150
 - 0s - loss: 1.0760 - acc: 0.3400
Epoch 5/150
 - 0s - loss: 1.0700 - acc: 0.3400
Epoch 6/150
 - 0s - loss: 1.0640 - acc: 0.3500
Epoch 7/150
 - 0s - loss: 1.0581 - acc: 0.3700
```

```
Epoch 8/150
 - 0s - loss: 1.0520 - acc: 0.4200
Epoch 9/150
 - 0s - loss: 1.0467 - acc: 0.5100
Epoch 10/150
...
```

Predicting new Unseen Data

Let's see how we did by predicting on **new data**. Our model has **never** seen the test data that we scaled previously. This process is the exact same process we would use on totally brand new data.

```
predict = model.predict(scaled_X_test)

classes = np.argmax(predict, axis=1)

print(classes)
```

```
array([1, 0, 2, 1, 2, 0, 1, 2, 2, 1, 2, 0, 0, 0, 0, 1, 2, 1, 1, 2, 0, 2,
       0, 2, 2, 2, 2, 2, 0, 0, 0, 0, 1, 0, 0, 2, 1, 0, 0, 0, 2, 1, 1, 0,
       0, 1, 2, 2, 1, 2], dtype=int64)
```

Evaluating Model Performance

These are our predictions of the classes of plant in our test data. Now we need to compare with our y_test data (test labels)

```
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score

print(confusion_matrix(y_test.argmax(axis=1), predictions))
```

```
array([[19,  0,  0],
       [ 0, 13,  2],
       [ 0,  0, 16]], dtype=int64)
```

```
print(classification_report(y_test.argmax(axis=1),predictions))
```

```
precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      0.87      0.93        15
           2       0.89      1.00      0.94        16

   micro avg       0.96      0.96      0.96        50
   macro avg       0.96      0.96      0.96        50
weighted avg       0.96      0.96      0.96        50
```

Here we can see we evaluated the model using the metrics precision, recall and f1-score.

Save and Load Model

```
model.save('iris-model.h5')

from keras.models import load_model

newIrisModel = load_model("iris-model.h5")
print(newIrisModel.predict_classes(X_test))
```

```
array([2, 1, 2, 2, 2, 1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 2, 2, 2, 2, 2, 1, 2,
       1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 2, 1, 1, 2, 2, 1, 1, 1, 2, 2, 2, 1,
       1, 2, 2, 2, 2, 2], dtype=int64)
```

3.23 Recurrent Neural Networks

One of the major issues with feed forward networks is that it didn’t allow us to have a many sequence input i.e. a review. We realised that feed forward networks trained on individual words in the text and this lost all context to any reviews we wanted to predict the sentiment of.

We learned that RNNs are specifically designed to work with sequence data e.g. time series data or sentences. A sentence is just a sequence of words.

A common example of using recurrent neural networks is sentiment scores. You can feed in a sequence of words, maybe a paragraph of a movie review, and then request back a vector indicating whether it was a positive or negative sentiment.

Initial thought process on how RNNs will work:

- Sequence input
- to a single vector value
- training data - bunch of paragraphs and then some sentiment score attached.
- train RNN on that data to have a sequence input to a vector output

Recurrent Neuron

A recurrent neuron is a little different to a normal neuron. It sends the output back to itself.

Over time, the neuron is receiving inputs from the current timestep but also from the previous timestep.

Cells that are a function of inputs from the previous timestep are also known as memory cells.

You could then begin to think that is has some form of memory because we're passing in historical data into that recurrent neuron.

A common example of using RNN for this sort of input and output is sentiment scores.

So you can feed in a sequence of words, maybe a paragraph of a movie review, and then request back a vector indicating whether it was a positive sentiment, such as they really like the movie (usually indicated by 1). Or they really dislike the movie (usually indicated by -1or 0).

We realised RNNs were exactly what we needed to use in order to implement our sentiment analysis functionality.

3.24 The Vanishing Gradient Problem

One of the major problems of RNN is the Vanishing gradient. In any neural network, the weights are updated in the training phase by calculating the error and back-propagation through the network. But in the case of RNN, it is quite complex because we need to propagate through time to these neurons. This can often lead to the gradient becoming close to zero which results in the weights changing very slightly and the training process can be very slow.

3.25 LSTM (Long Short Term Memory)

An issue RNN faces is that after awhile the network will begin to “forget” the first inputs, as information is lost at each step going through the RNN.

We need some sort of “long-term memory” for our networks. An LSTM has a similar control flow as a recurrent neural network. It processes data passing on information as it propagates forward. The differences are the operations within the LSTM’s cells. These operations are used to allow the LSTM to keep or forget information.

LSTM helps to solve the vanishing gradient problem. Gradients are used to update rnn parameters. For long sequences of words this gradient becomes smaller and smaller to the point where no network training can take place. LSTM help to overcome this problem and make it possible to capture long term dependencies in a sequence of words.

For example:

“I like to eat chocolates” - like and chocolates are close together

“I like, whenever there is a chance and usually there are many of them, to eat chocolates.”

- “like” and “chocolates” are separated by a big distance.

Long term dependencies between 2 words are handled much better with LSTM networks.

3.26 Django

- Django research - why we chose django (MVT, Layered architecture, ORM, Testing)

For this project we decided to use Python's Django web Framework. We didn't have any knowledge of Django and so some research had to be done before we fully committed to building our project.

Some of the key reasons why we chose to use Django:

- **Object relational mapping api (ORM)** - which allows us to write python classes to represent our database tables. This means we won't have to write sql queries to interact with our database.
- **Admin** - when we have written the classes to represent our data, django can generate an Admin interface. This is a web based user interface for editing content. This is generated automatically by django. This is a very powerful tool which can speed up the development cycle.
- Django allows **templates** to generate your web pages and **url mappings** to configure URLs for our web-app.
- It will also allow us to handle **html forms** with minimal code.
- Huge amount of 3rd party **packages** which can help
- Built in User Authentication
- HTTP session handling
- Powerful high performance caching system

Django Models

Allows us to create data and store that in a database.

Models:

- Are Python classes and their purpose is to make our data persistent i.e. they let us store things in a database in such a way that if you stop the Django server and then restart it the data will still be there.
- A model class is mapped to a database table. So a model class called user will result in a users table and a meeting class will have a meetings table.

Each object of that class can be stored in a row in that table. So if I create a new user and save that in the database the users table will contain a new row with the data for that user i.e user1 will have a row to itself, user2 will have a row to itself etc.

Migrations:

- A migration is a Python script that changes the database so as to keep the structure of the database up to date with our model classes.
- We might add or remove or rename properties. If I decide I want to store the user's age, I will add an age property to the user class. But that means that I need to add an age column to the database aswell**.** Every time we change our model class we will need to change the corresponding database table so that it will have the right columns to store our objects. That change to the database structure is automated through migrations.
- Django will generate a migration script that will update the database table accordingly. When the project gets advanced, django won't be able to generate everything and you will need to write your own migration code.

Django Views

The view is a Django component that handles a request (GET request or POST request) for a web page

```
from django.shortcuts import render
from django.http import HttpResponse


# Create your views here.

def welcome(request):
    return HttpResponse("Welcome to MyTrip!")
```

Django Templates

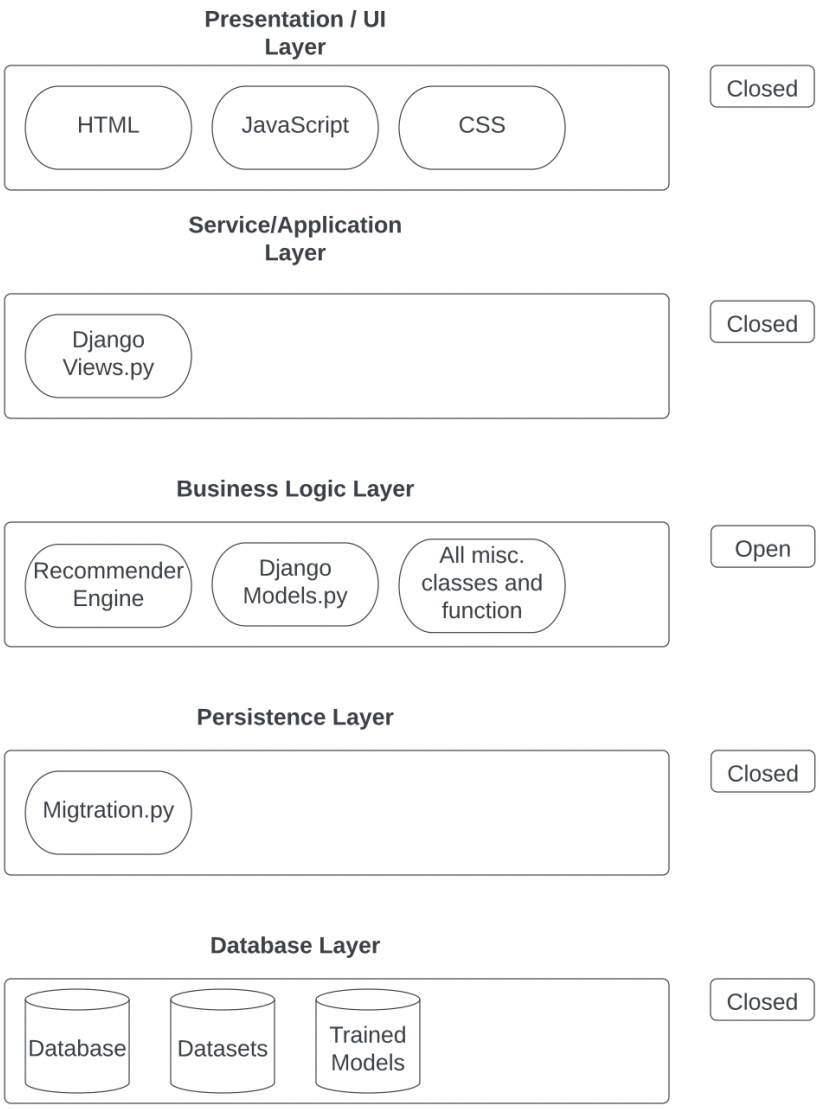
Templates are the components that are responsible for displaying our data to our user

Understanding flow of data

- Generate HTML
- Call template from view
- Pass data from view to template
- Pass data from template to view
- Insert to database

4. Design

4.1 Layered system architecture



We decided to use the layered system architecture approach as it is one of the most well-known software architecture patterns. This architecture pattern splits the code up into “layers”, where each layer has a certain responsibility and provides a service to a higher layer.

The layers included are:

- Presentation Layer
- Application Layer
- Business logic layer
- Persistence Layer

- Database Layer

Higher layers are dependent upon and make calls to the lower layers.

The presentation layer contains the HTML pages that will be displayed to the user and handles user interaction.

The application layer sits between the presentation layer and the business logic layer. It handles the input from the user through GET and POST requests.

The business logic layer is the engine of the web app. Here, we have our recommendation algorithms and models as well as our sentiment analysis algorithm and model. This layer also consists of our Django models.

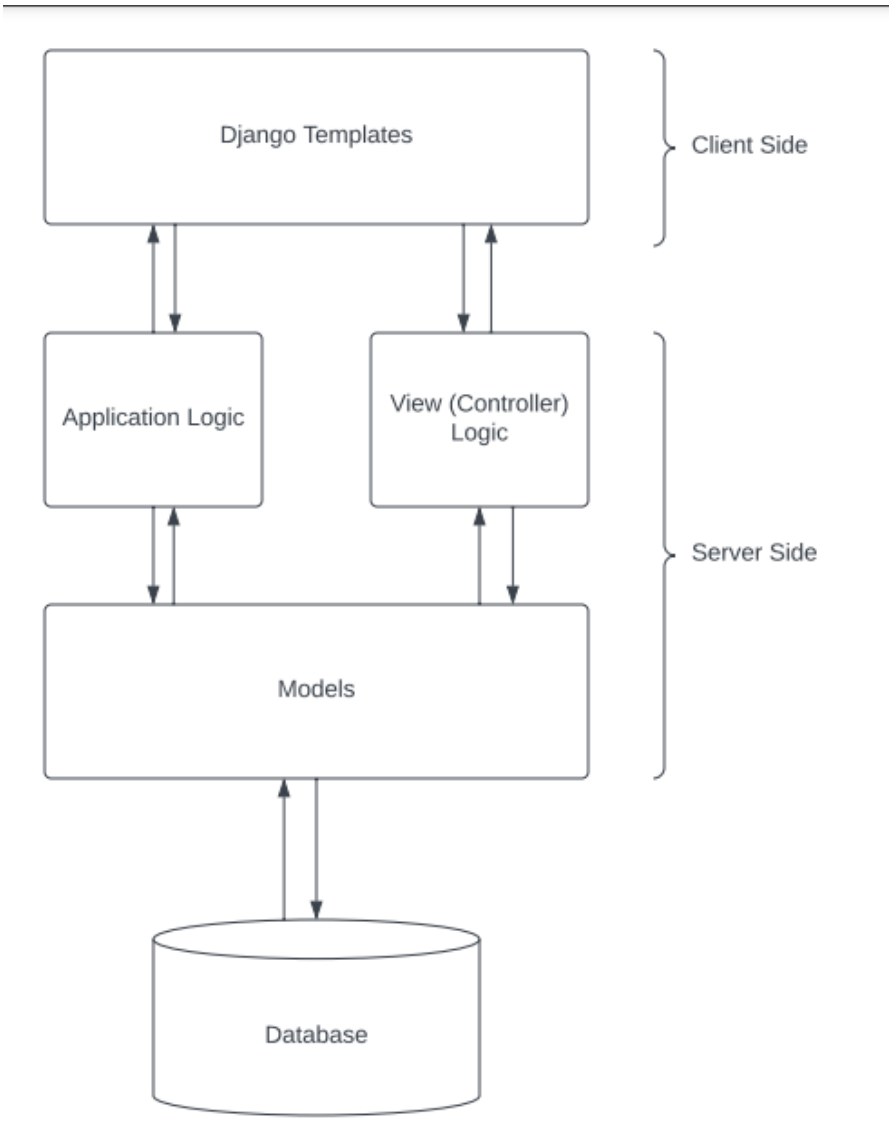
The persistence layer contains the code to access the database layer. It is used to manipulate the database in order to create tables or add columns etc.

The database layer is the underlying database (SQLite). This stores our user data such as user accounts, trips created, restaurant and activity ratings and reviews. It also allows us to store our attractions (restaurants and activities), attraction features and similar attractions that our machine learning algorithm predicts.

Layers that are closed means that a request must go through them while layers that are open means that a request can bypass this layer and go to the relevant layer after.

4.2 Django Architecture

- Django Architecture - diagram + explain



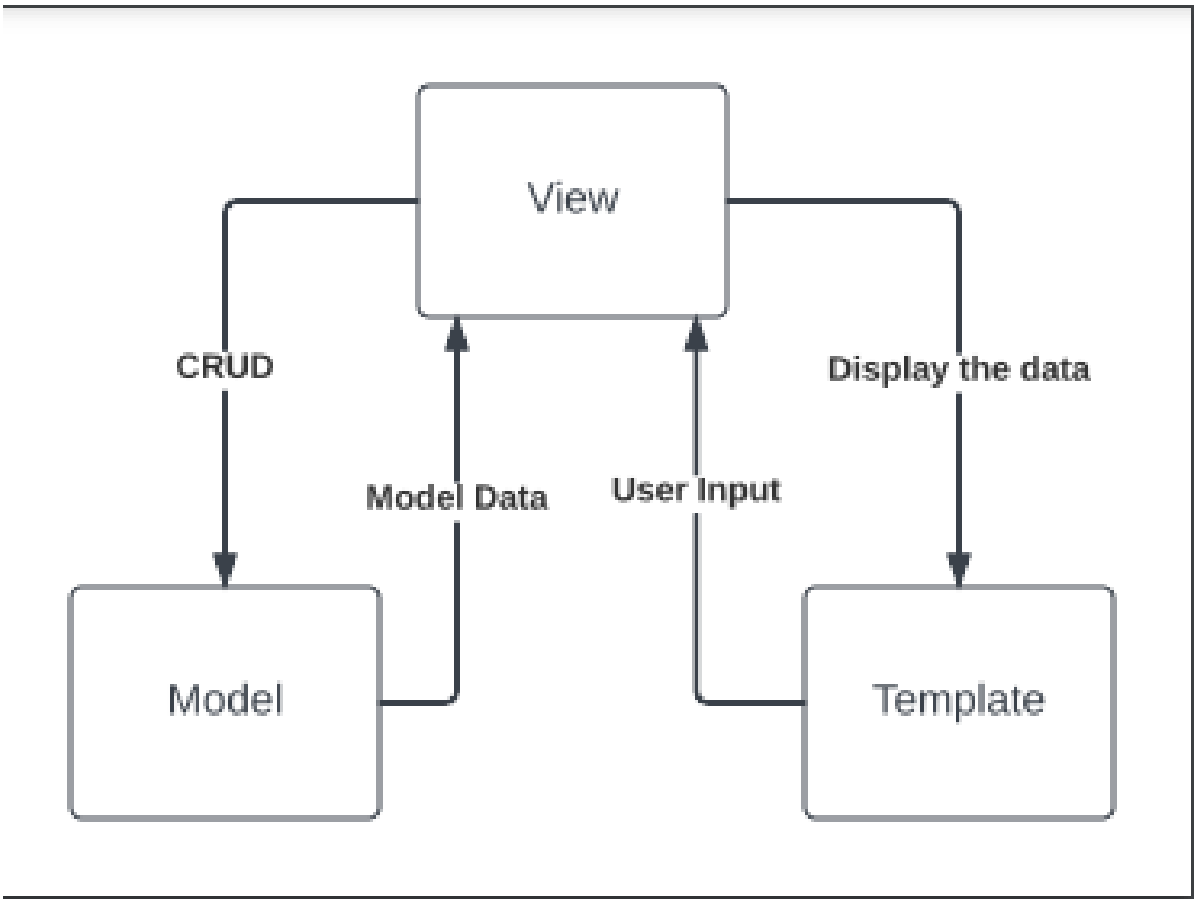
This is a further breakdown of our Django architecture. It outlines the client/server architecture and what is on both sides.

Client side: The presentation layer is on the client side where the user will interact with the web app and the templates will be rendered.

Server side: The server side consists of our business logic and Django models

4.3 MVT Architecture

- Django MVT (MVC) approach - diagram + explain



The MVT (Model View Template) software design pattern is Django's equivalent to MVC (Model View Controller). We used this design pattern as it emphasizes the separating of data representation from the other components that interact and process the data.

There are three components in the MVT pattern with each having a specific purpose:

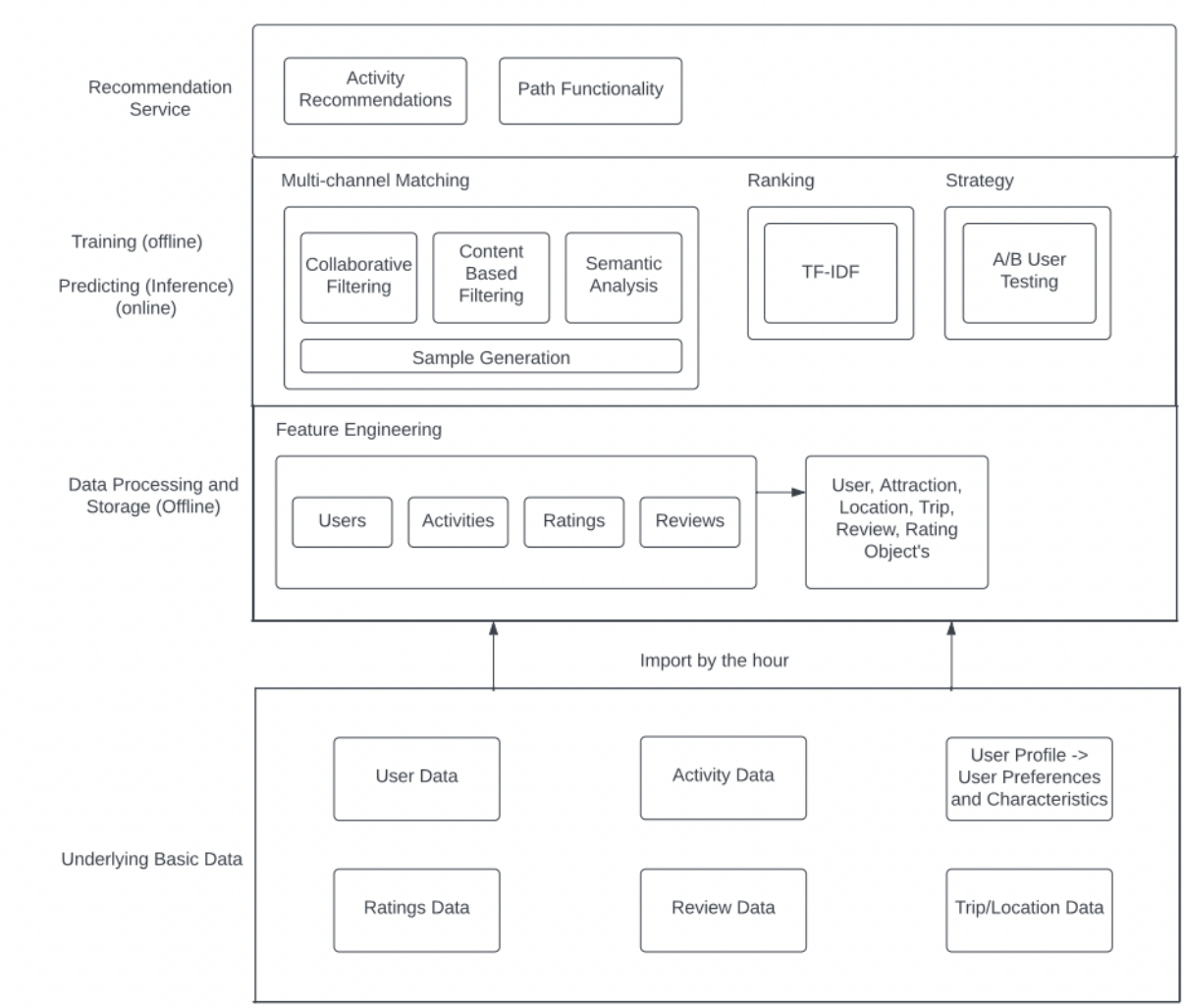
Model: The model is the source of information for our data. It acts as an interface for the data and is represented by our SQLite database.

View: The view handles HTTP requests and returns HTTP responses. It interacts with the model and renders the template. It also executes the business logic code.

Template: The templates acts as the presentation layer and which contains the HTML pages that will be displayed to the user.

4.4 Recommender Engine

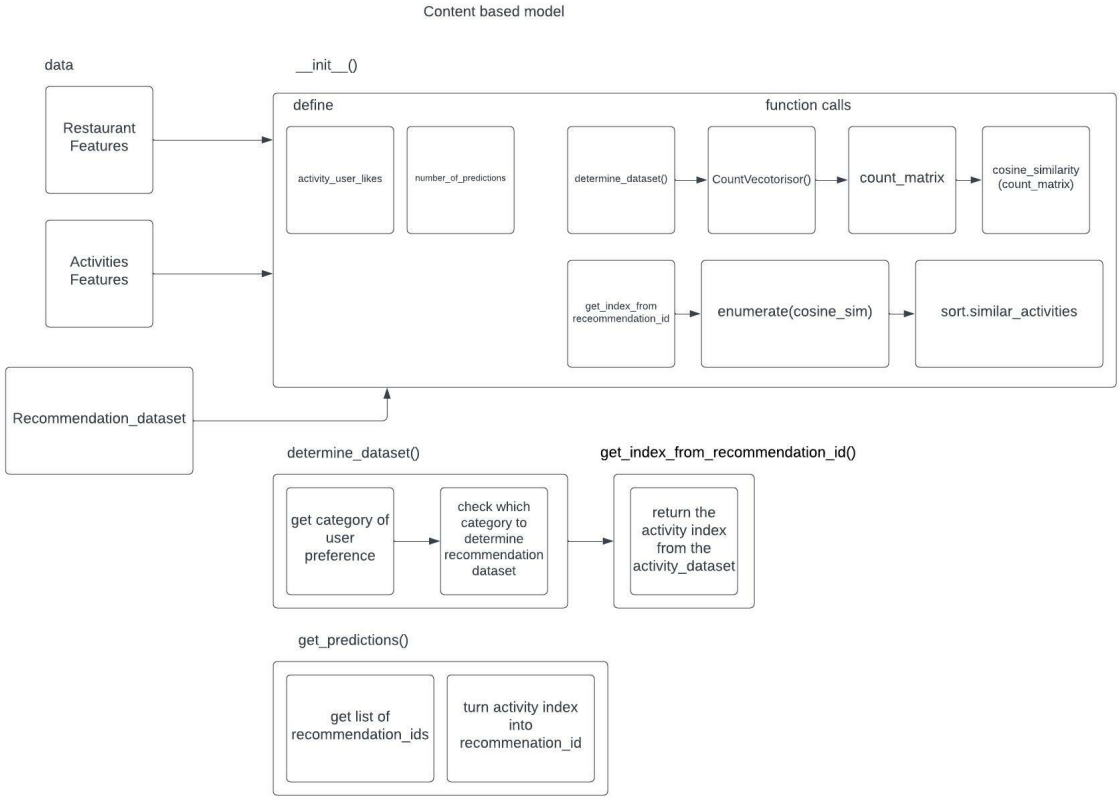
Below is the system architecture we planned to implement after our initial research. This built upon our plan of implementation from our functional spec. This was a guide throughout our design phase which coincided with our early implementation phase. We cover the functionality of each layer from the recommendation service to the database layer and everything in between.



4.5 Content Based Model

- The design of our content based recommender model changed drastically as we progressed throughout a project. The more we learned about the content based model, the more we learned about the requirements of our dataset.
- We began with the approach of a k-nearest neighbour model from our research. We started without using any major libraries and doing the calculations manually to

understand how the system worked. From this knowledge we decided that cosine similarity was our best option for designing the structure of our model. We used tfidf for a short time but found more consistent results using cosine similarity.



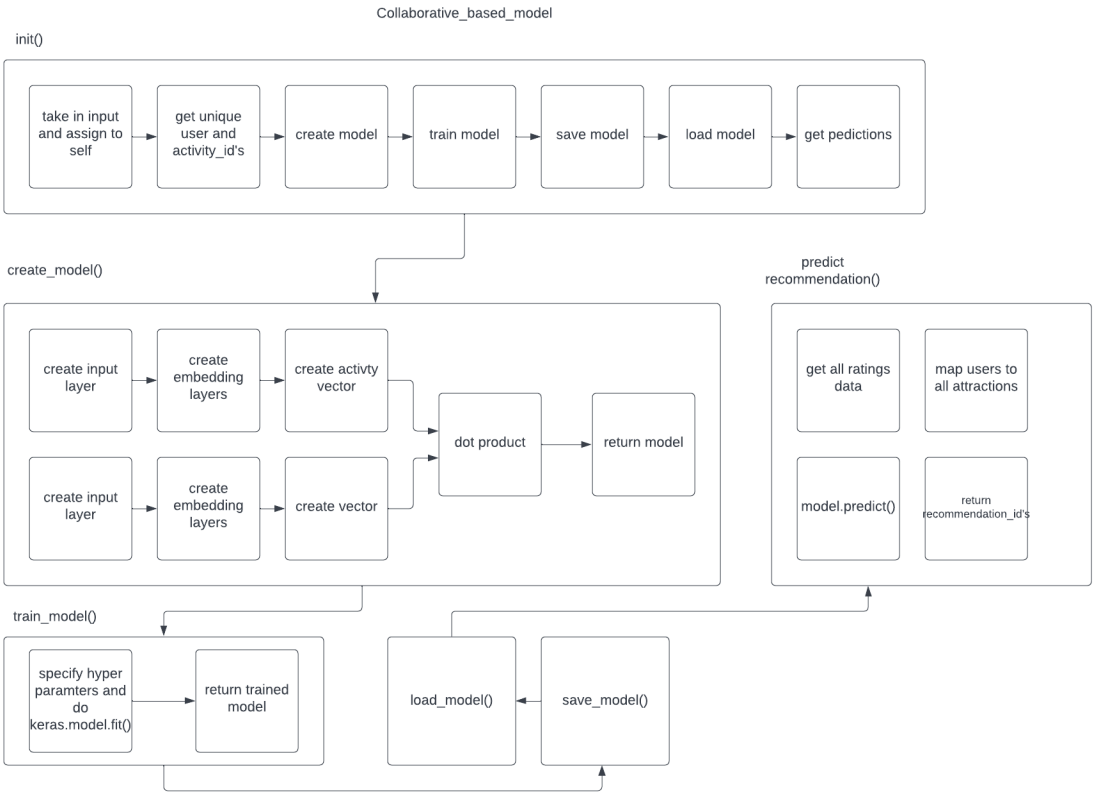
The flow of the content based model is stated above.

1. datasets
2. init functions (constructor)
    a. take in user input
        i. activity they would like recommendations for
        ii. user id
    b. determine dataset
    c. using count vectoriser
    d. using cosine similarity

    e. get index from rec_id
    f. sort by highest similarity score
3. determine dataset
4. get index from rec_df
5. sort the similar activities

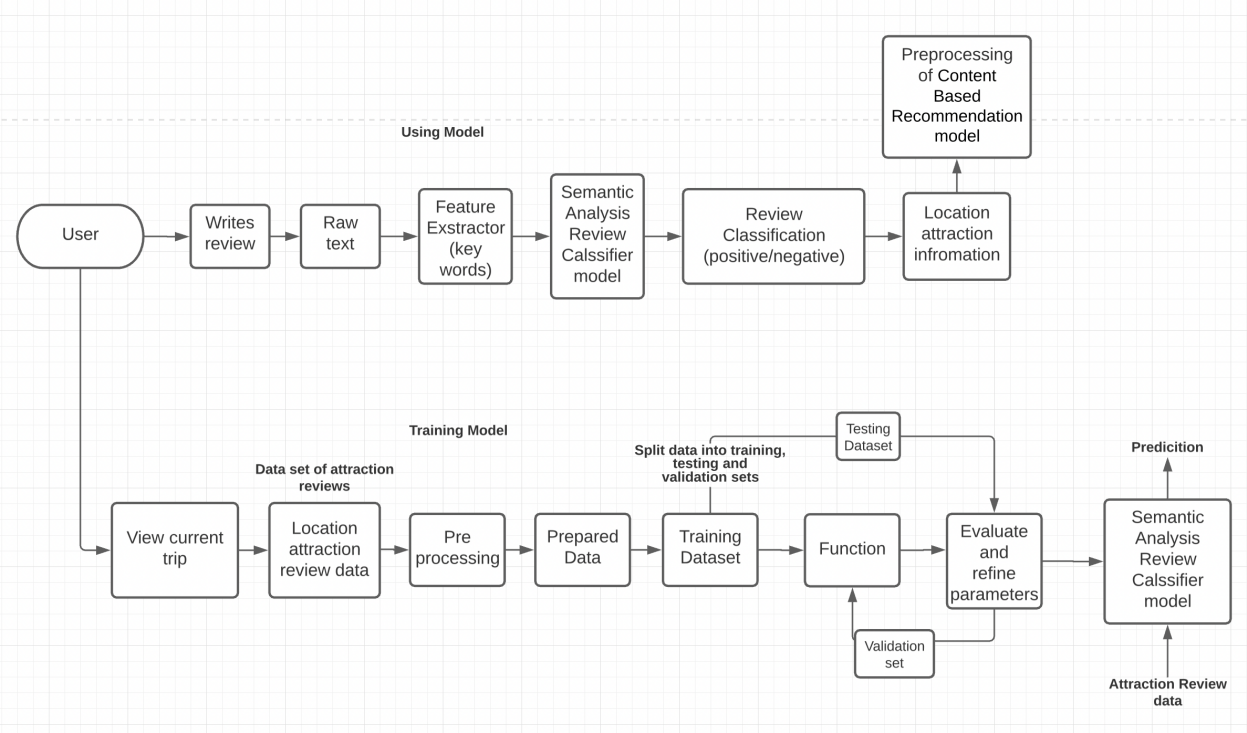
4.6 Collaborative Based Model

- our collaborative filtering model consists of the following functions
    1. init
    2. create model()
    3. train model()
    4. load_model()
    5. save_model()
    6. get_predictions()



- the model's functionality will be discussed below. When we were designing this model, we kept in mind what we learning in our research. A goal for the design was simple, yet functional code. This balance was the driving force behind our design.
- our own dataset

4.7 Sentiment Analysis



5. Implementation

5.1 Sentiment Analysis Implementation

The first step involved importing the necessary libraries needed for the implementation.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding
from tensorflow.keras.layers import LSTM
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import pandas as pd
import spacy
import re
from tensorflow import keras
import numpy as np
from ..models import Review
```

```
class SentimentAnalysis(object):

    def __init__(self):

        self.df = pd.read_csv(
        "webapp/sentiment_analysis/IMDBDataset.csv")  #To set up our initial sentiment analy

        self.nlp = spacy.load('en_core_web_md')  # using spacy to load in the language libra
        self.stopwords = self.nlp.Defaults.stop_words  # spacy has a list of stopwords built
```

We created a sentiment Analysis class to store all the relevant fields and methods associated with our sentiment analysis object once instantiated.

The first field of our sentiment analysis model, the IMDB dataset which contains 50,000 reviews in English with a sentiment associated with each review (positive/negative). We used the IMDB dataset because when analyzing reviews, we don't really care about the subject of the review, rather we care about the words used to describe it. We wanted to analyze if someone thought something was "great", "fantastic", or "bad" and "terrible". Whether the review was about a restaurant or a movie the same sentiment could still be applied. This cut down on the time needed to develop a completely new dataset and allowed us to focus on the recommendation datasets which we felt were more important.

We used Pandas throughout the project to read in our datasets(.csv files) and to manipulate data. Pandas is an open source Python package that is used in data analysis and machine learning tasks. It is built upon Numpy which supports multi-dimensional arrays. It allowed us to efficiently drop columns or null values in our dataset.

The 2nd field in our class was our English library of words. We used the Spacy library which is an open source Natural Language Processing library. It is designed to effectively handle NLP tasks with the most efficient implementation of common algorithms. Spacy is much faster and more efficient than NLTK, at the cost of the user not being able to choose algorithmic implementations.

The 3rd field is our list of stop-words. Spacy has a list of stop-words built in. We will use these in our preprocessing to eliminate any stop-words present in our dataset.

```
def preprocess_reviews_dataframe(self, df, review_field, preprocessed_review_field):
        df[preprocessed_review_field] = df[review_field].str.lower()
        df[preprocessed_review_field] = df[preprocessed_review_field].apply(
            lambda review: re.sub(r"(@[A-Za-z0-9]+)|([^0-9A-Za-z \t])|(\w+:\/\/\S+)|^rt|http

        df[preprocessed_review_field] = df[preprocessed_review_field].apply(lambda review: r

        df[preprocessed_review_field] = df[preprocessed_review_field].apply(
            lambda review: ' '.join([word for word in review.split() if word not in self.sto

        return df
```

One of the most important steps in building machine learning models is data preprocessing. In order to train the model well and for it to predict accurately, the first step is to normalise the dataset by converting everything into lower case.

We then use a regular expression to remove numbers, HTML tags and special characters.

Finally, one of the most important steps was removing stop-words. These words such as “is”, “of”, “the”, “for” don’t tell us anything about the sentiment and can result in overfitting of our model. We don’t want our model to memorise the dataset and so removing stop-words was crucial for our implementation.

The final preprocessed dataframe is then returned.

```
def binary_encode_sentiments(self, preprocessed_reviews_df):
        sentiment = {'positive': 1, #Creating a dictionary mapping the positive and negative
                     'negative': 0}

        preprocessed_reviews_df['sentiment'] = [sentiment[item] for item in preprocessed_rev

        return preprocessed_reviews_df
```

For RNNs to work, all text data has to be integer encoded before it is fed into the model. By creating a dictionary, we mapped the sentiments “positive” and “negative” to the integers 0 and 1 respectively. This was then used to convert all sentiments in our dataset to integers.

```
def tokenize_and_sequence_data(self, preprocessed_reviews_df):
        X = preprocessed_reviews_df['preprocessed_review'].values #Recurrent neural networks
        y = preprocessed_reviews_df['sentiment'].values #Converting the integer sentiments i
        #These are both numpy arrays

        #For RNNs to work, all text data has to be integer encoded before it is fed in to th
        tk = Tokenizer(lower=True) #The tokenizer has to be initialised. Here we are specify
        tk.fit_on_texts(X) #We fit the training data (preprocessed reviews stored in array X
        #Once the data is fit, the tokenizer can be used to encode our reviews as it maps un
        X_seq = tk.texts_to_sequences(X) #using text_to_sequences, we convert our tokens int
        X_pad = pad_sequences(X_seq, maxlen=80, padding='post') #We then pad the sequences t

        return X_pad, y, len(tk.word_counts.keys())+1
```

Recurrent neural networks require inputs in an array data type. Here we convert the preprocessed reviews into an array called X. We also store the integer sentiments in an array called y. These are both numpy arrays. We used Keras' built in tokenizer to integer encode our reviews. The tokenizer has to be initialised. Here we specify that it should use lower case words in order to ensure all text is normalised.

We fit the training data (preprocessed reviews that are stored in array X) on the tokenizer in order to tokenize our data into unique words. Once the data is fit, the tokenizer can be used to encode our reviews as it maps unique words to uniue integers.

Using text_to_sequence, we convert our tokens into a sequence of integers (where each integer represents a unique word).

We then pad the sequences to ensure all reviews have the same length. This is very important for RNNs. It will pad any sequence less than 80 integers with 0s. Any sequence greater than 80 will be shortened to integers.

```
def train_model(self, X_pad, y, vocab_size):
        #Split the data into train and test data using sklearn's train_test_split. the test
        X_train, X_test, y_train, y_test = train_test_split(X_pad, y, test_size=0.3, random_

        batch_size = 32
        vocabulary_size = vocab_size # embedding needs the size of the vocabulary. We get th
        max_words = 80 #Embedding needs the length of the input sequences (We used pad_seque
        embedding_size = 32 #This specifies the amount of dimensions that will be used to re
        model = Sequential() #Creating our model
        model.add(Embedding(vocabulary_size, embedding_size, input_length=max_words))
        model.add(LSTM(200, dropout=0.2, recurrent_dropout=0.2))#adding 1 hidden Long short
        model.add(Dense(1, activation='sigmoid')) #Use the sigmoid activation function in ou
```

```
        model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']) #A

        model.fit(X_train, y_train, batch_size=batch_size, epochs=15, verbose=2)

        model.save("test.h5") #save the model
```

```
def preprocess_user_review(self, user_review, tk):
        user_review_lower = user_review.lower()  # Converting the user review into lower cas
        # will need to add preprocessing steps here - regular expressions to catch any noise
        # For now, this will suffice
        user_review_denoised = " ".join([word for word in user_review_lower.split() if
                                         word not in self.stopwords])  # Removing stopwords
        list_of_user_review = []
        list_of_user_review.append(user_review_denoised)

        user_review_array = np.array(
            list_of_user_review)  # As RNNs need the input in an array, appending the user r
        review_seq = tk.texts_to_sequences(
            user_review_array)  # using text_to_sequences, we convert our tokens into a sequ
        review_pad = pad_sequences(review_seq, maxlen=80,
                                   padding='post')  # We then pad the sequences to ensure al
```

```
def predict_sentiment(self, review_pad):
        sentiment_analysis_model = keras.models.load_model(
            "webapp/sentiment_analysis/test-for-web-app.h5")  # Loading our model from the f
        user_review_prediction = sentiment_analysis_model.predict(review_pad)  # Predict the
        check_user_review_prediction = np.where(user_review_prediction > 0.5, 1,
                                                0)  # If it's greater than 0.5, it's positiv
        sentiment = check_user_review_prediction[0][
            0]  # contained in a list in a numpy array so have to access it like so

        return sentiment
```

```
def get_average_sentiment_rating(self, reviews):  # reviews is a list of Review objects from
        total_positives = 0
        total_reviews = 0

        for review in reviews:  # This is an O(n) approach, is there a faster more efficient
            if review.sentiment == 1:
```

```
                total_positives += 1
            total_reviews += 1

        average_rating = (total_positives / total_reviews) * 100
        average_rating = round(average_rating, 1)
        return average_rating
```

```
def store_user_review_in_database(self, request, attraction, user_review, sentiment):
        review = Review(attraction=attraction, user=request.user, review=user_review,
                        sentiment=sentiment)  # Creating a review object (have to specify wh
        review.save()  # Inserting into the database
```

5.2 Content Based Model Implementation

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

- we start by importing pandas for reading in our datasets used to build the count_matrices
- countVectoriser is imported as a way to extract features from activity vectors
- these vectors are then used in the cosine similarity

```
def __init__(self, activity_user_likes, number_of_predictions):

        print('intialising model...')
        print('')
        # self.restaurant_features = pd.read_csv('../recommendation_dataset/production_dataf
        # self.rec_df = pd.read_csv('../recommendation_dataset/production_dataframes/recomme

        self.activity_user_likes = activity_user_likes
        self.number_of_predictions = number_of_predictions

        self.rec_df = pd.read_csv('webapp/recommendation_dataset/production_dataframes/recom

        print('determining dataset...')
        self.determine_dataset()
```

```
        self.cv = CountVectorizer()
        print('building matrix...')
        self.count_matrix = self.cv.fit_transform(self.attraction_dataset['combined_features
        # print("Count Matrix: ",self.count_matrix.toarray())
        print('calculating cosine similarity...')
        cosine_sim = cosine_similarity(self.count_matrix)

        self.activity_index = self.get_index_from_recommendation_id()

        self.similar_activities = list(enumerate(cosine_sim[self.activity_index]))

        self.sorted_similar_activities = sorted(self.similar_activities, key = lambda x:x[1]

        self.sorted_similar_activities = self.sorted_similar_activities[:number_of_predictio
        print(self.sorted_similar_activities)
```

- our init command is where all our functions are used to either load or create and train a model. Either way, by the end of the init, you will have predictions to recommend a user.

1. assign user input to self so it can be used across the entire class
2. read in the recommendation dataset so we can match recommendation ids later in the get predictions function
3. determine which dataset to use
    a. we do this based on the category of the users preference, if the preference is a restaurant, we only want to recommend them restaurants, and then if its not a restaurant, we only want to recommend them activities for this given preference.
4. we then use a CountVectorizer(), we are turning a sentence into a matrix of tokens. We can then use this to calculate the similarity between activities. This will give an activity its ‘k-nearest neighbour’ like we discussed in our research.
5. we then calculate the cosine similarity of all the matrices we just creates and then rank them, by most similar to our users preference.
6. we then sort them using lambda. key = lambda x:x[1] will sort on the cosine similarity value, which is our goal.
7. then we use our ‘number_of_predictions’ to get our top n recommendations for the user

```
def determine_dataset(self):
        print(self.activity_user_likes)
        # check which category is associated with the activity the user likes
        category = self.rec_df[self.rec_df.recommendation_id == self.activity_user_likes]["c
        print(category)
        if category == 'restaurant':
            self.attraction_dataset = pd.read_csv('webapp/recommendation_dataset/production_
        else:
            self.attraction_dataset = pd.read_csv('webapp/recommendation_dataset/production_
```

- we do this based on the category of the users preference, if the preference is a restaurant, we only want to recommend them restaurants, and then if its not a restaurant, we only want to recommend them activities for this given preference.

```
def get_index_from_recommendation_id(self):
        # return self.rec_df[self.rec_df.activity_title == self.activity_user_likes]["index"
        # activity_index = self.attraction_dataset[self.attraction_dataset.recommendation_id
        activity_index = self.attraction_dataset.index[self.attraction_dataset['recommendati

        return activity_index
```

- we want to return the recommendation id that is relevant to the user, not the index of the dataset it was trained on

```
def get_predictions(self):
        print('making predictions...')
        predictions = self.sorted_similar_activities
        list_of_recommendation_ids = []
        i = 0
        while i < self.number_of_predictions:
            recommendation_id = self.attraction_dataset[self.attraction_dataset.index == (pr
            list_of_recommendation_ids.append(recommendation_id)
            i += 1
            # print(recommendation_id)
            combined_features = self.attraction_dataset.combined_features[self.attraction_da

        # print(list_of_recommendation_ids)
        return list_of_recommendation_ids
```

- return the recommendation id for a given activity recommendation
- we also performed ad hoc testing here with ensuring the combined features reflected our cosine similarity results

5.3 Collaborative Filtering Model Implementation

```
import pandas as pd
import keras
import numpy as np
import tensorflow as tf
from keras import Input
from keras.layers.embeddings import Embedding
from sklearn.model_selection import train_test_split
from keras.layers import Input, Embedding, Flatten, Dot, Dense
from keras.models import Model
```

- we use pandas to import the datasets
- we import keras for clean and functional model building
- we user keras’s many sub libraries such as layers and models
- import train test split from sklearn to divide our datasets into trainable data
- import numpy to efficiently work with our numpy

```
class collab_based_model(object):

    # create model

    def __init__(self, userId, number_of_predictions):

        # read in up to date ratings data
        self.ratings = pd.read_csv('webapp/recommendation_dataset/production_dataframes/fina
        self.train, self.test = train_test_split(self.ratings, test_size=0.2, random_state=4

        # gather unique user id's and activity id's
        self.number_of_users = self.ratings.userId.nunique()
        self.number_of_activities = self.ratings.activityId.nunique()

        # # apply the mapping
        self.userId = userId
```

```
        self.number_of_predictions = number_of_predictions

        self.model = keras.models.load_model("webapp/recommendation_engine/recommendation_mo
        # self.model = self.create_model()

        # self.train_model()


        self.recommended_activity_ids = self.predict_recommendations()
```

- our init command is where all our functions are used to either load or create and train a model. Either way, by the end of the init, you will have predictions to recommend a user.

1. assign user input to self so it can be used across the entire class
2. import our ratings dataset
3. load in our model if already trained
4. train model if not
5. get unique users and activities to avoid duplicate recommendations
6. get predictions

```
def create_model(self):
        activities_input = Input(shape=[1], name="Activity-Input")
        activities_embedding = Embedding(self.number_of_activities+1, 5, name="Activity-Embe
        activities_vec = Flatten(name="Flatten-Activities")(activities_embedding)

        user_input = Input(shape=[1], name="User-Input")
        user_embedding = Embedding(self.number_of_users+1, 5, name="User-Embedding")(user_in
        user_vec = Flatten(name="Flatten-Users")(user_embedding)

        # hidden layers
        x = Dense(128, activation='relu')(activities_vec)
        x = Dense(64, activation='relu')(x)
        x = Dense(32, activation='relu')(x)

        # output layer
        activities_vec = Dense(1)(x)

        # hidden layers
        y = Dense(128, activation='relu')(user_vec)
        y = Dense(64, activation='relu')(y)
        y = Dense(32, activation='relu')(y)
```

```
        # output layer
        user_vec = Dense(1)(y)

        prod = Dot(name="Dot-Product", axes=1)([activities_vec, user_vec])
        model = Model([user_input, activities_input], prod)
        model.compile('adam', 'mean_squared_error')

        return model
```

- we create an input layer using keras
- we then create the embedding for both the user matrix and the activity matrix
- these are then burned into flatten vectored
- specify our hidden layer parameters
    - number of neurons
    - activation function
- specify the output layers
- using the product to multiply our resulting vectors together to get our resulting matrix factorisation results.
- we can then use this to get predicitons

```
def train_model(self):
    history = self.model.fit([self.train.userId, self.train.activityId], self.train.ratings,

    # save the model
def save_model(self):
    self.model.save("recommender_model.h5")

 def load_model(self):
    self.model = keras.models.load_model("webapp//recommendation_engine/recommendation_model
```

- the options are to either train and save the model, or to load an existing model

```
def predict_recommendations(self):
        activity_data = np.array(list(set(self.ratings.activityId)))
        user2allattractions = np.array([self.userId for i in range(len(activity_data))])
        predictions = self.model.predict([user2allattractions, activity_data])
```

```
        predictions = np.array([a[0] for a in predictions])
        recommended_activity_ids = (-predictions).argsort()[:self.number_of_predictions]
        # print(recommended_activity_ids)
        recommended_activity_ids = recommended_activity_ids.tolist()
        return recommended_activity_ids
```

- return the recommendation id for a given activity recommendation
- this is done with the following steps
    1. take in the ratings data from the dataset
    2. map the user id to all attractions
    3. get predictions for the user against all attractions using the model.predict()
    4. sort the recommendations and take the top n based on number of predictions
    5. convert to list so we can send to the front end for the use

6. Problems Solved

- Preprocessing to prevent overfitting - don’t want it memorising the dataset
    - for this we found sklearn’s implementation of train test split which prepared out data into sections for training our model.
- Content based - problem + solution
    - the quality of the content based model was proportional to the quality of the dataset. Since most tutorials we could find were working with pre-processed data, we had to find out our models behaviour with our dataset and how our data could be improved, which we discuss below.
- Collaborative filtering - problem + solution
    - the big problem with collaborative filtering is that it is difficult to test. The only metric for validation after MSE is user testing. This was difficult due to the fact that it relies on the dataset being really reliable and accurate. We provided our best solution to this by implementing a bias to ratings based on a users likes. So now if a user has been identified as similar to another through our model based on their ratings, it has more validity and the user can trust it more.

- Datasets - getting the right data and amount of data (size)
    - we were unable to find any existing dataset that suited our needs. This was a multi week search until we eventually realised that we were going to have to essentially build our own. This involved a large amount of pre-processing and re-factoring all the way up until the deadline for the project. As we learned more about our system and its needs, our datasets evolved constantly.
- Splitting of datasets
    - originally, the datasets was one entity. Activities and restaurants shared a space. This meant that when a user preference was passed to the recommender, it would get back a list of similar attractions and restaurants, which is not what we wanted. We then split this dataset into two. In our content based model, the model decides which dataset to use based on what category the given attraction is. This leads to focused recommendations that are more accurate and a better user experience.
- Research - getting an understanding of machine learning
    - when we started this project we had no understanding of machine learning or any neural networks. This is not part of our college course. We had to start from the beginning and continue to build on our knowledge.
    - The solution to this were plural-sight and uDemy courses that were mentioned in the research.
- The similar activities were too similar - not diverse enough - had to make the recommendations more diverse - adding more features to the activity_features dataset
    - originally our datasets included two
- working remotely
    - since there were several covid spikes throughout the semester, most of the work was done remotely.
- gitlab issues / lack of version control
    - for the last 3 weeks of our exam we were plagued with gitlab and version control issues which resulted in losing portions of code and being unable to push/pull/clone from the repo for a period of time. Within the last two weeks of

the project we had nearly no version control at all, so our productivity was halved right as we were about to start our final implementation. This cost us very valuable time. Our solution to this was paired programming which we have undertaken for the last two weeks of the project.

- Data in ratings matrix was sparse
  - our original ratings dataset/matrix was over 13GB which is unusable in our implementation. It contains large amounts of sparse data.
  - The solution to this was to build a ratings dataset that mapped a user, to an activity they rated and the activity. So now we have no sparse data. This improved the performance of our code dramatically.
- Time contraints/other modules
  - Both of us were focused on getting high grades this semester and that includes other modules that we had. We had to divide our time between all four modules including the project to ensure we get a high average grade in our final year.
- Content based recommendations load time
  - for each recommendation that user likes, we would produce a list of similar attractions. This was costly in terms of time and computational cost.
  - We implemented a solution by storing the similar attractions in the database
  - this meant that the database would just have to be queried with a computational complexity of n(1). It was n squared before the solution was implemented.

7. Results

7.1 Unit Testing

Our unit testing consisted of testing all URLs to ensure we got a response code 200. We wanted to ensure the code to our URLs worked correctly and we weren’t getting any 500 response - internal server error.

```
class TestUrls(TestCase):
#Testing all our endpoints
  def test_register_url(self):
          url = reverse('register') #get the url for register
          self.assertEquals(resolve(url).func, registerPage)
```

```
            #assert the register url function is registerPage

        def test_sign_in_url(self):
            url = reverse('sign-in')
            self.assertEquals(resolve(url).func, sign_in)

        def test_home_url(self):
            url = reverse('home')
            self.assertEquals(resolve(url).func, home)

        def test_user_interests_url(self):
            url = reverse('user-interests')
            self.assertEquals(resolve(url).func, user_interests)

        def test_create_trip_url(self):
            url = reverse('create-trip')
            self.assertEquals(resolve(url).func, create_trip)

        def test_logout(self):
            url = reverse('logout')
            self.assertEquals(resolve(url).func, logoutUser)

        def test_current_trip(self):
            url = reverse('current-trip')
            self.assertEquals(resolve(url).func, current_trip)
```

We also unit tested to ensure our views that didn’t require a login worked as expected too.

```
class TestViews(TestCase):
    def test_register_GET_request(self):

        response = self.client.get(reverse('register')) #GET the register page

        self.assertEquals(response.status_code, 200)
        self.assertTemplateUsed(response, 'webapp/register.html') #Assert the register html

    def test_sign_in_GET_request(self):

        response = self.client.get(reverse('sign-in'))

        self.assertEquals(response.status_code, 200)
        self.assertTemplateUsed(response, 'webapp/sign-in.html')
```

We also unit tested all functions that were part of our Models.

```
def test_get_rating_and_attraction_id(self):
        rating_and_attraction_id = "4+12"
        rating_value, attraction_id = Rating.split_rating_and_attraction_id(rating_and_attra
        self.assertEquals(rating_value, 4)
        self.assertEquals(attraction_id, 12)
```

7.2 Integration Testing

We tried to include as many integration tests but due to time constraints we didn’t get to write integration tests to cover the entirety of our code.

```
class TestUser(TestCase):

    def create_user(self):
        user = User.objects.create_user(username="test1", email="test@gmail.com", password="
        user.save()
        return user

    def create_super_user(self):
        super_user = User.objects.create_superuser(username="supertest1", email="supertest@g
        return super_user

    def test_User(self):
        user = self.create_user()
        self.assertEquals(user.username, "test1")
        self.assertEquals(user.email, "test@gmail.com")
        self.assertTrue(user.is_active)
        self.assertFalse(user.is_staff)
        self.assertFalse(user.is_superuser)

    def test_super_user(self):
        super_user = self.create_super_user()
        self.assertEquals(super_user.username, "supertest1")
        self.assertEquals(super_user.email, "supertest@gmail.com")
        self.assertTrue(super_user.is_active)
        self.assertTrue(super_user.is_staff)
        self.assertTrue(super_user.is_superuser)

    def test_create_user(self):
        with self.assertRaises(TypeError):
            User.objects.create_user()
        with self.assertRaises(ValueError):
            User.objects.create_user(username="")
        with self.assertRaises(ValueError):
            User.objects.create_user(username="", email="test@gmail.com")
        with self.assertRaises(ValueError):
            User.objects.create_user(username="", email="test@gmail.com")
```

We created integration tests to ensure we could create user accounts. These would then be saved in the database and allow us to proceed testing with the rest of the web app.

Once we had a test user account in our database, we tested the login functionality:

```
def test_sign_in_and_access_home_page(self):
        #self.client.post()
        TestUser().create_user()
        self.client.login(username="test1", password="test2022")
        response = self.client.get(reverse("home"))
        self.assertTemplateUsed(response, 'webapp/index.html')
```

The user-interests page:

```
def test_user_interests_get_request(self): #At the moment the test case doesn't have any att
        #Need to create attractions
        attraction_model = TestAttractionModel()
        attraction_model.create_attraction_objects()
        test_user = TestUser()
        test_user.create_user()
        self.client.login(username="test1", password="test2022")
        response = self.client.get(reverse('user-interests'))
        self.assertEquals(response.status_code, 200)
```

User viewing an event/attraction

```
def test_event_get_request(self):
        attraction_model = TestAttractionModel()
        attraction_model.create_attraction_objects()
        test_user = TestUser()
        test_user.create_user()
        self.client.login(username="test1", password="test2022")
        response = self.client.get(reverse('view_event', kwargs={'id': 2}))
        self.assertEquals(response.status_code, 200)
```

We also created integration tests for all our Django Models such as TestAttractionModel, TestReviewModel, TestTripModel, TestRatingModel, TestUser,

TestUserPreferenceModel, TestLocationModel. We wanted to ensure all the data could be stored in our database and retrieved from the database too.

```
def test_attraction(self):
        self.create_attraction_objects()
        attraction = Attraction.objects.get(pk=1)
        activity_title = attraction.get_activity_title()
        address = attraction.get_address()
        category = attraction.get_category()
        latitude = attraction.get_latitude()
        longitude = attraction.get_longitude()
        details = attraction.get_details()
        average_rating = attraction.get_average_rating()
        self.assertEquals(activity_title, "Spoons")
        self.assertEquals(address, "London City")
        self.assertEquals(category, "restaurant")
        self.assertEquals(latitude, 1234554.0093)
        self.assertEquals(longitude, 123454325.653)
        self.assertEquals(average_rating, 60.0)
        self.assertEquals(details, "details")
```

7.3 Git + Ci/CD usage

Throughout this entire project we made use of Git for version control. This involved regular pushes and pulls from our repo to ensure we were always on the same version when developing.

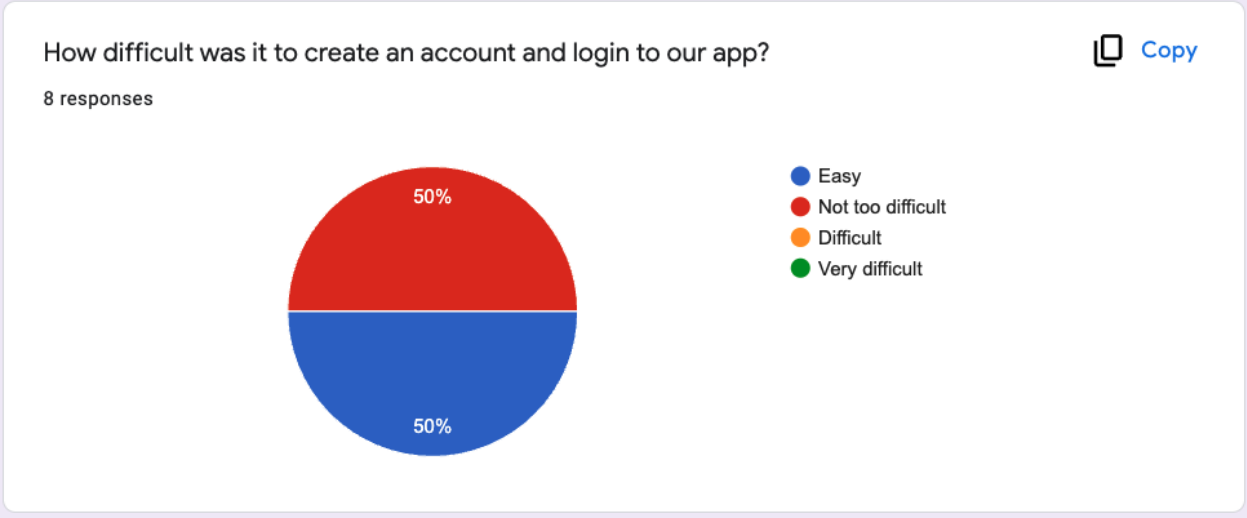
We also tried to make use of Gitlab's continuous integration functionality to run our test suite with every commit. Unfortunately we couldn't get this fully working due to some problems with Git. During the development process we accidentally committed a large dataset which broke our repo to an extent.

7.4 Adhoc Testing

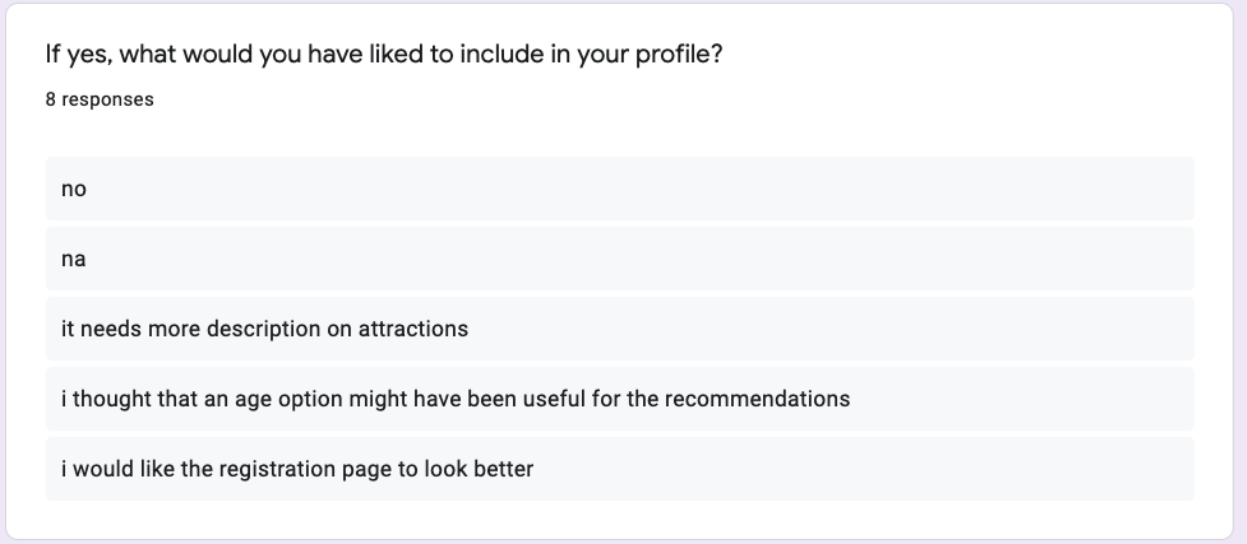
- A piece of our testing was ad hoc. This means that the testing had no documentation or planning. Since we were constantly experimenting with features and implementations of components that we were unsure if they would be final. Features were added and were testing using such adhoc methods of testing such as python print statements or analysing outputs by hand. An example of this was when calculating similar scores using cosine, we would also print combined features (uncomment print statement to view this test in content based model). By printing this we were able to see if the activities were truly similar and also what criteria in the count matrix lead to a cosine similarity score for x.
- Print statements were used more than any other adhoc method. This meant that we were able to see what the system was doing at any given state. We also used vscode debugging to step through code and identify bugs.

7.5 User Testing

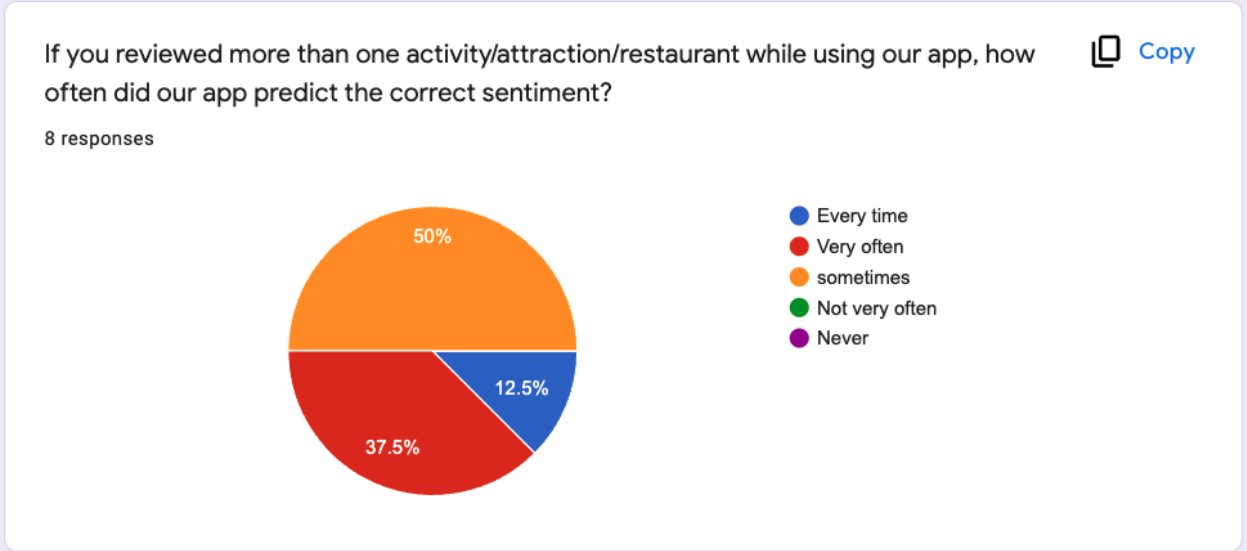
We performed user testing on 8 candidates. The first half were conducted in a group while the second half were conducted individually. We asked a serious of questions after the user had followed a path through the system. Below we have documented some of the most important pieces of information we learned from user testing.



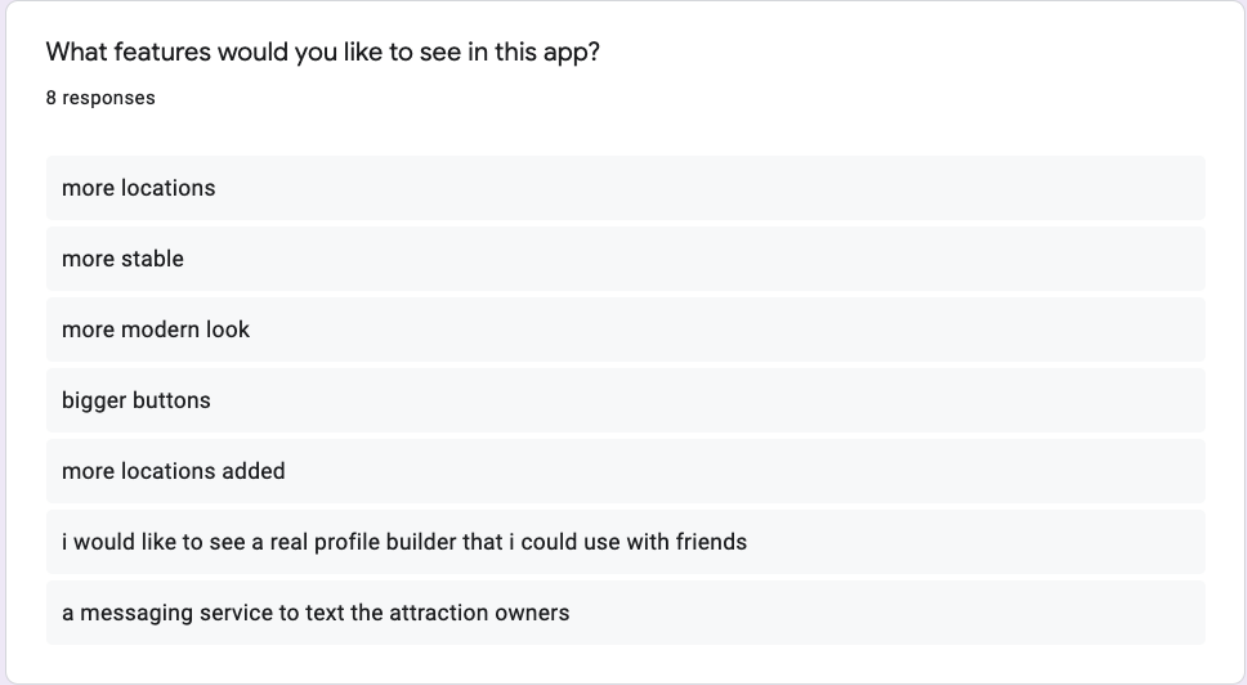
The users seemed to be divided on the create account page. When users were asked to comment on this, they believed that the create a count and log in page needed work in design. With this feedback we used bootstrap css to design a more friendly user interface. This is now implemented in our code.



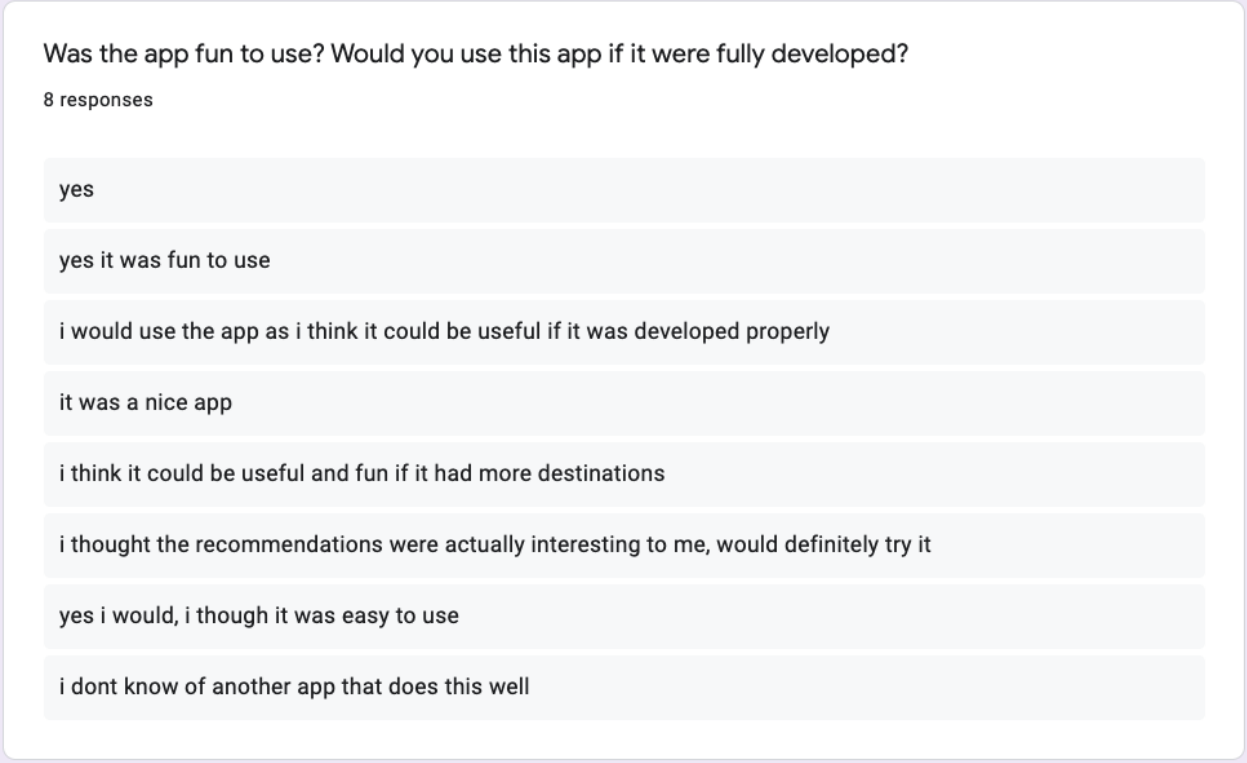
When asked what users would like to add to their profile, we noticed one user saying that our attractions needed more descriptions. As a result of this we added the category and combined features to the attraction to ensure the user could make a more informed decision.



Some said that it got the correct sentiment. Some said that it could have been better. We addressed this by continuing to refine parameters to ensure the users have a more accurate result.



The biggest features that the users wanted to see in the app were more locations and a better UI experience in terms of the look.



Overall, it seemed that the application was fun to use and people would be interested at a version of the app that was developed further.

The user testing was successful. It brought our attention to features that we had not considered. We were so focused on the implementation of the models, we neglected the front end user experience. We have now addressed this matter and the application is better due to the user testing.

8. Future work

- We would like to develop this app further based on some of the features mentioned in the user testing.
- We would like to expand this application to more locations.
- We would also like to add javascript to make a more interactive experience rather than just using bootstrap.
- optimising models with scheduler
- improving dataset quality
- asynchronous processing

    - always have recommendations ready for the user, so there is no loading times
- have established datasets