**Content**:

* Major Dependencies
* Inheritance & User-defined method
* Encapsulation
* Accessing & Manipulating several databases
* Sending Email

**Recommendation Algorithm:**

* 2D Arrays & ArrayLists
* Quadruple For loops for Searching (Linear Search)
* Cosine Similarity
* Predictions
* Selection Sort

**Major Dependencies**

**Backend:**

* Firebase Realtime Database, Firebase Storage & Firebase Authentication: Used to access their respective databases and login, display products, and recommend products (see **Accessing & Manipulating several databases** below for more detail and see Crit\_A\_Planning)
* Internet Permission: Asks for permission to use the internet to access databases to display products, otherwise user cannot login or register
* External Storage Permission: Used to require permission to read external storage when U is creating a product and picking an image for the product
* Filter: Used to filter products based on searched products in “Shop” using the **performFiltering()** and **publishResults()** methods[[1]](#footnote-0)
* Uri: When picking an image in the “Backend” screen, Uri stores the location of it on the device which is necessary to display the image of the product

**Frontend:**

* Picasso: Finds a products image by its url and displays it[[2]](#footnote-1)
* Date: When a product is purchased, the date of purchase is included in the email, notifying U of the products purchase
* ProgressBar: Displays a progress bar while products are being loaded in to clarify to the user that the product is not buggy, just loading
* Bitmap: Gets the bits of images to display in the “Shop” and “Product Display” screens
* Android Studio GUI: This includes buttons, texts, and more, which are necessary to display and receive information to and from the user

**Inheritance & User-defined method**

The **MyAdapter** class extends its child class **MyViewHolder** as seen below.

public class **MyAdapter** extends **RecyclerView**.**Adapter**<**MyAdapter**.**MyViewHolder**>

This inheritance is necessary to display the products in the “Shop” screen because **MyViewHolder** contains all of the details of each product and the **onBindViewHolder()** and **onCreateViewHolder()** methods use this functionality to display the details of each product appropriately as seen below.

// Creates a ViewHolder with all of the content  
@NonNull  
@Override  
public MyViewHolder **onCreateViewHolder**(@NonNull ViewGroup parent, int viewType) {  
 LayoutInflater inflater = LayoutInflater.from(context);  
 View view = inflater.inflate(R.layout.my\_row, parent,false);  
 return new MyViewHolder(view);  
}  
  
// For each item in ViewHolder, run the following method  
@Override  
public void **onBindViewHolder**(@NonNull MyViewHolder holder, @SuppressLint("RecyclerView") int position) {  
 holder.text1.setText(Data.get(position).getTitle());  
 holder.text2.setText(Data.get(position).getDescription());  
 holder.text3.setText("₹" + Data.get(position).getPrice());  
  
 //Setting image  
 Picasso.get()  
 .load(Data.get(position).getPictureUrl())  
 .placeholder(R.mipmap.ic\_launcher)  
 .fit()  
 .centerCrop()  
 .into(holder.image);  
  
 holder.cl.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void **onClick**(View view) {  
 //Sending clicked product to ProductDisplay  
 Intent i = new Intent(context, ProductDisplay.class);  
 i.putExtra("data1", Data.get(position).getTitle());  
 i.putExtra("data2", Data.get(position).getDescription());  
 i.putExtra("price", String.valueOf(Data.get(position).getPrice()));  
 i.putExtra("image", Data.get(position).getPictureUrl());  
 context.startActivity(i);  
 }  
 });  
}  
  
@Override  
public int **getItemCount**() {  
 return Data.size();  
}  
  
public class **MyViewHolder** extends **RecyclerView**.**ViewHolder** {  
 TextView text1, text2, text3;  
 ImageView image;  
  
 ConstraintLayout cl;  
  
 public **MyViewHolder** (@NonNull View viewItem) {  
 super(viewItem);  
 text1 = viewItem.findViewById(R.id.Title\_txt);  
 text2 = viewItem.findViewById(R.id.Description\_txt);  
 text3 = viewItem.findViewById(R.id.PriceDisplay);  
 image = viewItem.findViewById(R.id.TempImage);  
 cl = viewItem.findViewById(R.id.mainLayout);  
 }  
}

**onBindViewHolder()** is also a user-defined method as the **onClick()** method inside only runs when a user presses the product. It is necessary for this method to be user-defined, as otherwise there is no way of determining whether the **ProductDisplay** class should run and then the user cannot buy any products.

**Encapsulation**

| public class **BackendDataStorage** {  private String Title, Description;  private int Price;  private String PictureUrl;   public **BackendDataStorage** () {   }   public **BackendDataStorage**(String title, String description, int price, String pictureUrl) {  Title = title;  Description = description;  Price = price;  PictureUrl = pictureUrl;  }   public String **getTitle**() {  return Title;  }   public String **getDescription**() {  return Description;  }   public int **getPrice**() {  return Price;  }   public String **getPictureUrl**() {  return PictureUrl;  }   public void **setTitle**(String title) {  Title = title;  }   public void **setDescription**(String description) {  Description = description;  }   public void **setPrice**(int price) {  Price = price;  }   public void **setPictureUrl**(String pictureUrl) {  PictureUrl = pictureUrl;  } } |
| --- |

The **BackendDataStorage** class above encapsulates the title, description, price, and image url of each product. This is necessary to get and set the title, description, price, and image url variables of a specific product that is desired throughout the program. This also avoids accidentally changing a variable by mistake. For example, **MyAdapter** has an ArrayList **Data** containing objects of the **BackendDataStorage** class. The **onBindViewHolder()** method uses the getters of the **BackendDataStorage** class to set the details of each product in the “Shop” screen, as seen in the “**Inheritance & User-defined method**” section above. Without encapsulation, this would not be possible. Also, redundancy is reduced and future extensibility is easier due to the organization of code.

**Accessing & Manipulating several databases**

| From Backend.java  ------------------  // Sends item to database public void **SendData** (View view) {  // Getting & converting values  String title = ((EditText)findViewById(R.id.Title)).getText().toString().trim();  String description = ((EditText)findViewById(R.id.Description)).getText().toString().trim();  String priceStr = ((EditText)findViewById(R.id.Price)).getText().toString();   // If image, title, description, or price not not, do not continue  if (BF == null || title.isEmpty() || description.isEmpty() || priceStr.isEmpty()) {  Toast.makeText(Backend.this, "Must include image, title, description, and price!", Toast.LENGTH\_LONG).show();  return; }  int price = Integer.parseInt(priceStr);  // Top node of the database  rootNode = FirebaseDatabase.getInstance();  reference = rootNode.getReference("Items");   // Storing image on Firebase Cloud  StorageReference fileReference = StorageRef.child(title  + "." + getFileExtension(SelectedImage));   // Putting image in Firebase Storage & Firebase Realtime Database  fileReference.putFile(SelectedImage).continueWithTask(new Continuation<UploadTask.TaskSnapshot, Task<Uri>>() {  @Override  public Task<Uri> **then**(@NonNull Task<UploadTask.TaskSnapshot> task) throws Exception {  if (!task.isSuccessful()) {  throw task.getException();  }  return fileReference.getDownloadUrl();  }  }).addOnCompleteListener(new OnCompleteListener<Uri>() {  @Override  public void **onComplete**(@NonNull Task<Uri> task) {  if (task.isSuccessful()) {  Toast.makeText(Backend.this, "File Uploaded!", Toast.LENGTH\_SHORT).show();  BackendDataStorage BDS = new BackendDataStorage(title, description, price, task.getResult().toString());  DatabaseRef.child(title).setValue(BDS);  }  else {  Toast.makeText(Backend.this, "Task failed: " + task.getException().getMessage(), Toast.LENGTH\_LONG).show();  }  }  }); }  public String **getFileExtension** (Uri uri) {  ContentResolver CR = getContentResolver();  MimeTypeMap mime = MimeTypeMap.getSingleton();  return mime.getExtensionFromMimeType(CR.getType(uri)); }  From ProductDisplay.java  ------------------------  public void **Send\_Rating** (View view) {  if (RB.getRating() != 0) {  // Add rating to firebase  String UserUID = FirebaseAuth.getInstance().getCurrentUser().getUid();  DatabaseRef.child(UserUID).child(data1).setValue(RB.getRating());  Toast.makeText(this, "Rating Sent!", Toast.LENGTH\_SHORT).show();  }  else {  Toast.makeText(this, "Rating cannot be 0", Toast.LENGTH\_SHORT).show();  } } |
| --- |
|  |

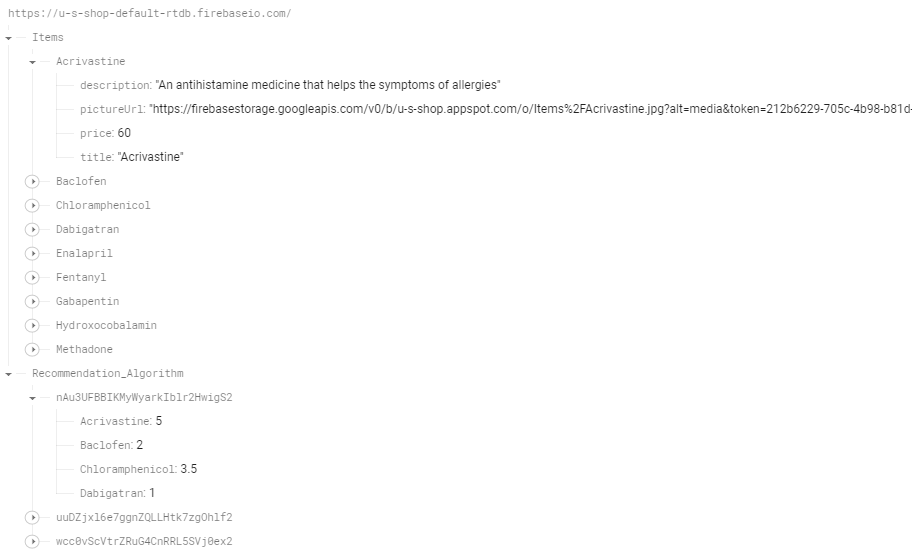
From Register.java

------------------

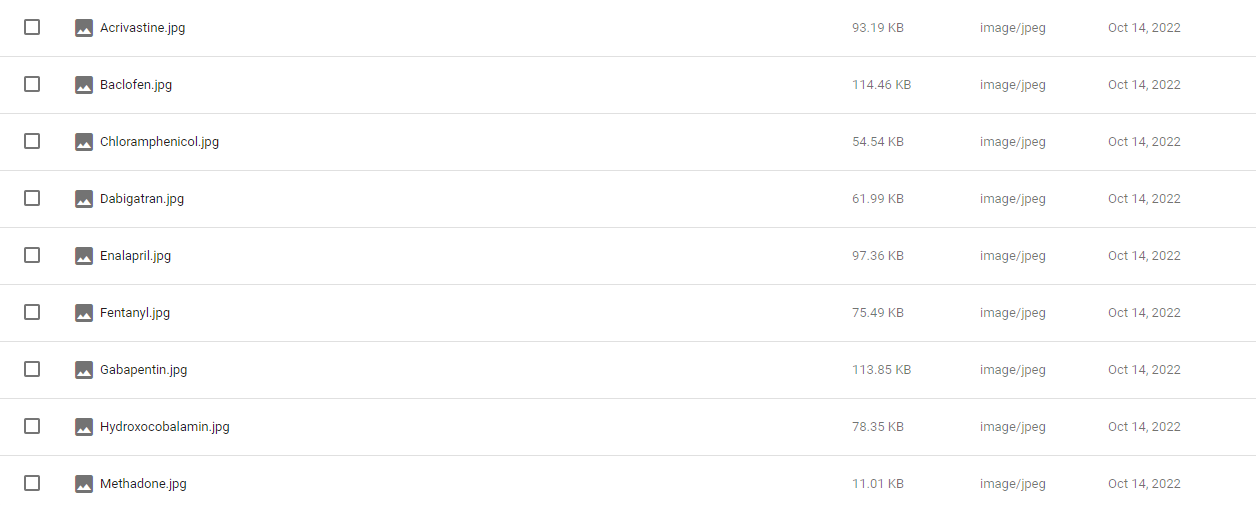
public void **CreateAccount** (View view) {  
 // If not in U's location, do not create account  
 String PINInput = PIN.getText().toString();  
  
 if (!PINInput.equalsIgnoreCase("12345")) {  
 findViewById(R.id.Warning).setVisibility(View.VISIBLE);  
 return;  
 }  
  
  
 FirstName = ((EditText)findViewById(R.id.EnteredFirstName)).getText().toString().trim();  
 LastName = ((EditText)findViewById(R.id.EnterLastName)).getText().toString().trim();

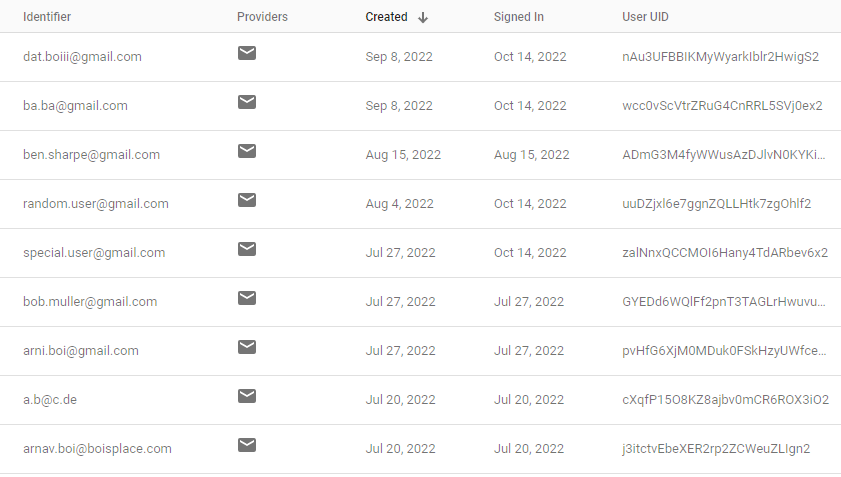
// Ensure '.' or '@' is not used anywhere in first or last name  
if (FirstName.contains(".") || LastName.contains(".") || FirstName.contains("@") || LastName.contains("@")) {  
 Toast.makeText(Register.this, "The first nor the last name can contain '.' or '@'", Toast.LENGTH\_SHORT).show();  
 return;  
}  
  
 String email = FirstName + "." + LastName + "@gmail.com";  
 String password = Password.getText().toString().trim();  
  
 // Creating Account & checking if it was successful  
 MainActivity.fAuth.createUserWithEmailAndPassword(email, password).addOnCompleteListener(new OnCompleteListener<AuthResult>() {  
 @Override  
 public void **onComplete**(@NonNull Task<AuthResult> task) {  
 if (task.isSuccessful()) {  
 Toast.makeText(Register.this, "Account Created!", Toast.LENGTH\_SHORT).show();  
 startActivity(new Intent(getApplicationContext(), Shop.class));  
 }  
 else {  
 Toast.makeText(Register.this, "Error: " + task.getException().getMessage(), Toast.LENGTH\_SHORT).show();  
 }  
 }  
 });  
}

By nesting methods, the above code sends the details of each product, each user's ratings, and each account created to FRD, FS, and FA. This is achieved using user-defined methods. They are required to receive input from the user when a product is created, rated, or an account is created, so the information can be sent to the appropriate database. Below shows how the databases store this data.

**Firebase Realtime Database:**

**Firebase Storage:**



**Firebase Authentication:**

From Login.java

----------------

| public void **LoggingIn** (View view) {  FirstName = ((EditText)findViewById(R.id.EnteredFirstName)).getText().toString().trim();  LastName = ((EditText)findViewById(R.id.EnteredLastName)).getText().toString().trim();   Email = FirstName + "." + LastName + "@gmail.com";  Password = ((EditText)findViewById(R.id.EnteredPassword)).getText().toString().trim();   // Signing In  MainActivity.fAuth.signInWithEmailAndPassword(Email, Password).addOnCompleteListener(new OnCompleteListener<AuthResult>() {  @Override  public void **onComplete**(@NonNull Task<AuthResult> task) {  if (task.isSuccessful()) {  // Checking if user is U. Else loads screen for the Shop  if (Email.equalsIgnoreCase("special.user@gmail.com") && Password.equals("1a9+7\*")) {  Toast.makeText(Login.this, "Special Account Login Successful!", Toast.LENGTH\_SHORT).show();  startActivity(new Intent(getApplicationContext(), Backend.class));  return;  }   Toast.makeText(Login.this, "Login Successful!", Toast.LENGTH\_SHORT).show();  startActivity(new Intent(getApplicationContext(), Shop.class));  }  else {  Toast.makeText(Login.this, "Error: " + task.getException().getMessage(), Toast.LENGTH\_SHORT).show();  }  }  }); } |
| --- |

FA loops through all users’ account details to search whether an account matches. If a match is found and the details of U’s account are entered, then the user is sent to the “Product Creation” screen, else they are sent to the “Shop” screen. Otherwise, the user is notified that no account matches. The products in FRD are used in the **onBindViewHolder()** method to display the products on the “Shop” screen, as seen in the “**Inheritance & User-defined methods**” section above. FRD uses FS to display the images of the products. The ratings in FRD are used by the recommendation algorithm to restructure the order of the products[[3]](#footnote-2). This shows that databases are important to the application, as otherwise accounts, products, and a recommendation algorithm would not exist.

**Sending Email**

From ProductDisplay.java

-------------------------

| //Runs if "Buy" button is clicked public void **Purchased** (View view) {  //Getting Users First & Last Name  String NameMail = MainActivity.fAuth.getCurrentUser().getEmail(), FirstName = "", LastName = "";   // Getting First & Last Name from the users mail  boolean ReachedDot = false, ReachedAtSymbol = false;  Date CurrentTime = Calendar.getInstance().getTime();  for (int MailIndex = 0; MailIndex < NameMail.length(); MailIndex++) {  if (!ReachedDot) {  if (NameMail.charAt(MailIndex) != '.') {  FirstName += NameMail.charAt(MailIndex);  }  else {  ReachedDot = true;  }  }  else if (!ReachedAtSymbol) {  if (NameMail.charAt(MailIndex) != '@') {  LastName += NameMail.charAt(MailIndex);  }  else {  ReachedAtSymbol = true;  }  }  }   // Sending Mail  String SendMailto = "arnav.kumarltp@gmail.com", Subject = "<<BUSINESS>> " + FirstName + " bought " + Title.getText().toString(),  Message = FirstName + " " + LastName + " on " + CurrentTime.toString() + " bought " + Title.getText().toString() + " for ₹" + Price;   JavaMailAPI JMAPI = new JavaMailAPI(this, SendMailto, Subject, Message);  JMAPI.execute(); }  public class **JavaMailAPI** extends **AsyncTask**<**Void**,**Void**,**Void**> {  . . .  protected Void **doInBackground**(Void... params) {  //Creating properties  Properties props = new Properties();   //Configuring properties for gmail  props.put("mail.smtp.host", "smtp.gmail.com");  props.put("mail.smtp.socketFactory.port", "465");  props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");  props.put("mail.smtp.auth", "true");  props.put("mail.smtp.port", "465");   //Creating a new session  mSession = Session.getDefaultInstance(props,  new javax.mail.Authenticator() {  //Authenticating the password  protected PasswordAuthentication **getPasswordAuthentication**() {  return new PasswordAuthentication(Utils.EMAIL, Utils.PASSWORD);  }  });   try {  //Creating MimeMessage object  MimeMessage mm = new MimeMessage(mSession);   //Setting sender address  mm.setFrom(new InternetAddress(Utils.EMAIL));  //Adding receiver  mm.addRecipient(Message.RecipientType.TO, new InternetAddress(mEmail));  //Adding subject  mm.setSubject(mSubject);  //Adding message  mm.setText(mMessage);  //Sending email  Transport.send(mm);  } catch (MessagingException e) {  Toast.makeText(mContext,"Product can not be delivered",Toast.LENGTH\_SHORT).show();  }  return null;  } } |
| --- |

**Purchased()** uses nested if statements in a For loop to separate the current users first and last name from **NameMail** which extracts the users email by importing Firebase. This is necessary to inform U who to send the product to. **doInBackground()** accesses the internet and gmail by configuring the appropriate properties and sends an email in a try and catch statement. This is necessary to inform U of who has purchased a product, and the try and catch statement informs the user when this has failed. **JavaMailAPI** also inherits **AsyncTask** to run a thread of the purchase. This is necessary to not freeze the UI while sending the email[[4]](#footnote-3).

**Recommendation Algorithm**

**2D Arrays & ArrayLists**

From Recommendation\_Algorithm.java

----------------------------------

| // Setting Variables AllProducts = TempProducts; TempProductAmount = AllProducts.size(); DR = FirebaseDatabase.getInstance().getReference("Recommendation\_Algorithm");  // Fill other necessary variables DR.addValueEventListener(new ValueEventListener() {  @Override  public void **onDataChange**(@NonNull DataSnapshot snapshot) {  //Empty future used variables to reset  AllUsers.clear();  TempCounter = 0;   for (DataSnapshot UsersSnapshot : snapshot.getChildren()) {  // Adding all users  AllUsers.add(UsersSnapshot.getKey());   // Getting UserIndex  if (UsersSnapshot.getKey().equals(FirebaseAuth.getInstance().getCurrentUser().getUid())) {  UserIndex = TempCounter;  }  TempCounter++;  }  TempUserAmount = AllUsers.size();  Ratings = new double[TempUserAmount][TempProductAmount + 1]; // Row 0 is for Similarity |
| --- |

The **Recommendation\_Algorithm** class uses the two ArrayLists **AllUsers** and **AllProducts** and the 2D array **Ratings**. **AllUsers** contains the UiD of each user and **AllProducts** contains the name of each product in FRD. **Ratings** contains each user's rating of each product where the column is the user, corresponding to the index of **AllUsers**,and the row is the product, corresponding to the index of **AllProducts**, beginning at row 1. Row 0 is used for the similarity of each user's ratings of the products[[5]](#footnote-4).

ArrayLists were chosen over arrays as they are dynamic, allowing for the maximum number of users and products to change, which is important if the recommendation algorithm is run several times and those values have changed. Using the **AllUsers** and **AllProducts** ArrayLists enabled me to use the static data structure of 2D arrays which is more efficient than dynamic ones.

**Quadruple For loops for Searching (Linear Search)**

From Recommendation\_Algorithm.java

----------------------------------

for (DataSnapshot Users : snapshot.getChildren()) {  
 for (DataSnapshot Products : Users.getChildren()) {  
 // For each product, look through the 'AllUsers' & 'AllProducts' 2D arrays  
 // to get the index of the rating  
 for (int user = 0; user < AllUsers.size(); user++) {  
 for (int product = 0; product < AllProducts.size(); product++) {  
 if (Users.getKey().equals(AllUsers.get(user)) && Products.getKey().equals(AllProducts.get(product).getTitle())) {  
 Ratings[user][product + 1] = Products.getValue(Double.class);  
 }  
 }  
 }  
 }  
}

The above quadruple For loop loops through each product for each user in FRD and linearly searches each index in **AllProducts** and **AllUsers** till a match is found for every user's rating of each product. This is necessary to find all the correct indexes and values of **Ratings**. This certainly could be more efficient. However, U has at most 100 users with 50 products, which results in efficiency, which average devices can run in 2 seconds[[6]](#footnote-5). Hence, I decided that the inefficiency of the algorithm does not pose a significant problem for U’s desires and was kept. This can be modified later if it becomes a problem.

**Cosine Similarity**

| // Getting similarity of current user to all other users using cosine similarity public static void **Get\_Cosine\_Similarity** () {  // Cosine Similarity is (x . y) / (||x|| \* ||y||) where  // x . y is dot product of vectors x and y  // ||x|| \* ||y|| is the cross product of the length of vectors x and y   // Getting ||x||  for (int i = 1; i < Ratings[UserIndex].length; i++) {  UserLength += Math.pow(Ratings[UserIndex][i], 2);   if (Ratings[UserIndex][i] == 0) {  AllUnratedRows.add(i);  }  }  UserLength = Math.sqrt(UserLength);   boolean Exclude = false;  // Getting similarity for each user  for (int User = 0; User < Ratings.length; User++) {  // Resetting variables for each new user  TempDotProduct = 0; TempOtherUserLength = 0;   // Going through each users ratings & getting cosine similarity  for (int Rating = 1; Rating < Ratings[User].length; Rating++) {  if (User != UserIndex && Ratings[User][Rating] != 0) {  // Dot Product for each other user  TempDotProduct += Ratings[UserIndex][Rating] \* Ratings[User][Rating];   // Step 1 of getting ||y||  TempOtherUserLength += Math.pow(Ratings[User][Rating], 2);  Exclude = false;  }  }  // Step 2 of getting ||y||  TempOtherUserLength = Math.sqrt(TempOtherUserLength);   // ||x|| \* ||y||  double CrossProduct = UserLength \* TempOtherUserLength;   // Getting cosine similarity for current indexed user & setting it  double Cosine\_Similarity = TempDotProduct / CrossProduct;  Ratings[User][0] = Cosine\_Similarity;  } } |
| --- |

Cosine similarity is a famous mathematical equation that can be used to measure the similarity between different users’ ratings to the current user from 0 to 1; 1 is perfectly similar, 0 is opposite. This is necessary to predict the ratings of unrated products[[7]](#footnote-6). The formula and an example using the formula below shows how cosine similarity is calculated[[8]](#footnote-7).

**Formula:**

Where:

The similarity between users and

Current users rating of product

Other users rating of product

The number of products

**Before Similarity:**

|  | Similarity | Product 1 | Product 2 | Product 3 | Product 4 |
| --- | --- | --- | --- | --- | --- |
| User 1 |  | 1 | 5 | 3 | 0 |
| User 2 |  | 1.5 | 4.5 | 3 | 4 |
| User 3 |  | 5 | 1.5 | 1 | 1 |

**Calculating Similarities:**

**After Similarity:**

|  | Similarity | Product 1 | Product 2 | Product 3 | Product 4 |
| --- | --- | --- | --- | --- | --- |
| User 1 |  | 1 | 5 | 3 | 0 |
| User 2 | 0.81 | 1.5 | 4.5 | 3 | 4 |
| User 3 | 0.48 | 5 | 1.5 | 1 | 1 |

**Predictions**

From Recommendation\_Algorithm.java

----------------------------------

| // Predicting scores for unknown values public static void **Predictions** () {  for (int i = 1; i < Ratings[UserIndex].length; i++) {  if (Ratings[UserIndex][i] == 0) {  // Resetting values for each new empty rating  SimilaritySum = 0;  SimilarityDotProduct = 0;   // Get dot product of all other users with similarity score & rating in respective column  // Also getting Sum of similarities  for (int j = 0; j < Ratings.length; j++) {  if (j != UserIndex || Ratings[j][i] != 0) {  SimilaritySum += Ratings[j][0];  SimilarityDotProduct += Ratings[j][0] \* Ratings[j][i];  }  }  Ratings[UserIndex][i] = SimilarityDotProduct / SimilaritySum;  }  } } |
| --- |

The above algorithm linearly searches all rows for the current users ratings in **Ratings** till a value of 0 is found, meaning that a product is unrated. Dividing the dot product of all other users’ rating of that product and similarity by the sum of the similarities is a common method to get the predicted rating[[9]](#footnote-8). The predicted rating is necessary to order all products from highest ratings to lowest. Below is an example of User 1’s predicted rating of Product 4 in the “**After Similarity**” table above.

**Example:**

**Selection Sort**

From Recommendation\_Algorithm.java

----------------------------------

| // Getting all of current users' Ratings & Predicted Ratings double[] TempRatings = new double[TempProductAmount]; for (int i = 0; i < (Ratings[UserIndex].length - 1); i++) {  TempRatings[i] = Ratings[UserIndex][i + 1]; } Selection\_Sort(TempRatings);  . . .  public static void **Selection\_Sort** (double[] arr) {  for (int i = 0; i < arr.length - 1; i++)  {  int index = i;  for (int j = i + 1; j < arr.length; j++){  if (arr[j] > arr[index]){  index = j;//searching for greatest index  }  }  // Swapping temporary array  double biggerNumber = arr[index];  arr[index] = arr[i];  arr[i] = biggerNumber;   // Swapping Products  BackendDataStorage tempProduct = AllProducts.get(index);  AllProducts.set(index, AllProducts.get(i));  AllProducts.set(i, tempProduct);  } } |
| --- |

Now that all of the current users ratings have either been rated or predicted, the products just need to be sorted from highest to lowest according to the current users ratings. Selection sort was used to do so. Although selection sort is not the most efficient algorithm, U generally has at most 50 products, resulting in efficiency, which average devices can run in milliseconds[[10]](#footnote-9). If the inefficiency is later deemed a significant issue, it can be modified.

1. <https://www.youtube.com/watch?v=M73Vec1oieM> [↑](#footnote-ref-0)
2. <https://square.github.io/picasso/> [↑](#footnote-ref-1)
3. see “**Recommendation\_Algorithm**” section below for more detail [↑](#footnote-ref-2)
4. <https://developer.android.com/reference/android/os/AsyncTask> [↑](#footnote-ref-3)
5. in **Cosine Similarity** below, see **After Similarity** for visualization of **Ratings** [↑](#footnote-ref-4)
6. <https://www.visualcapitalist.com/millions-lines-of-code/> [↑](#footnote-ref-5)
7. see **Predictions** below for more detail [↑](#footnote-ref-6)
8. User 1 is current user [↑](#footnote-ref-7)
9. <https://www.youtube.com/watch?v=Fmtorg_dmM0> [↑](#footnote-ref-8)
10. <https://www.visualcapitalist.com/millions-lines-of-code/> [↑](#footnote-ref-9)