Università Politecnica delle Marche

FACOLTA' DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica e dell'Automazione

INTEGRAZIONE DI TECNICHE DI COMPUTER VISION E INTELLIGENZA ARTIFICIALE NELL'APP FITTRACKER

INTEGRATION OF COMPUTER VISION AND ARTIFICIAL INTELLIGENCE TECHNIQUES WITHIN THE FITTRACKER APPLICATION

Relatore:
Chiar.mo Prof.
SIMONE FIORI

Correlatore:
Prof.
EMANUELE STORTI

Presentata da:
ALESSANDRO RONGONI

Sessione Straordinaria
Anno Accademico 2021/22

Indice

Introduzione

L'interesse crescente per il fitness e il benessere personale ha portato allo sviluppo di applicazioni mobile avanzate che facilitano il monitoraggio e l'analisi dell'attività fisica. L'uso delle tecniche di computer vision e machine learning può offrire soluzioni avanzate per la rilevazione e il conteggio delle ripetizioni degli esercizi, nonché la stima dell'Indice di Massa Corporea (IBM) in tempo reale.

Questa tesi di laurea si concentra sull'utilizzo delle API di Firebase ML Kit per la computer vision e l'implementazione di modelli di machine learning all'interno di un'applicazione mobile per il fitness. L'obiettivo principale della ricerca è stato quello di addestrare modelli di machine learning utilizzando gli algoritmi di TensorFlow Lite e di integrarli all'interno dell'applicazione mobile per utilizzarli in tempo reale attraverso la fotocamera dello smartphone. In questo modo, è possibile rilevare la posa corporea dell'utente e contare le ripetizioni degli esercizi come push-ups e squat, nonché stimare l'IBM dell'utente in tempo reale.

La tesi di laurea presenterà l'analisi delle tecniche di computer vision e machine learning utilizzate, con particolare attenzione alla metodologia di addestramento dei modelli e all'integrazione degli stessi all'interno dell'applicazione mobile.

L'obiettivo finale è quello di valutare l'efficacia del sistema e le sue limitazioni attraverso una serie di test sperimentali condotti su un gruppo di utenti. La tesi di laurea presenterà i risultati ottenuti dal modello di machine learning, valutando la sua precisione e la sua usabilità.

I primi tre capitoli sono centrati sull'introduzione e la creazione dell'applicazione stessa, specificando i requisiti, la tipologia di architettura e lo sviluppo delle funzionalità di base di quest'ultima.
Dal quarto capitolo in poi verranno illustrate le tecniche, software e dataset utilizzati per introdurre all'interno dell'applicazione le funzionalità più avanzate come: l'assistente vocale, il pose detector e il BMI estimator. Per queste ultime due sono state utilizzate due diverse API appartenenti a Firebase ML Kit.
Infine, vengono esposte alcune considerazioni personali sui risultati ottenuti e su futuri miglioramenti.

1 Parte I: Introduzione a FitTracker

FitTracker è un'applicazione mobile creata da Federico Pretini, Alessandro Rongoni e Gregorio Vecchiola come progetto di esame per il corso di Programmazione Mobile presso l'Università Politecnica delle Marche. L'applicazione ha lo scopo di tenere traccia della dieta dell'utente e del suo benessere fisico e mentale.

L'applicazione permette infatti di annotare giornalmente quello che mangia l'utente e l'attività fisica svolta dallo stesso per permettergli di monitorare i progressi verso gli obiettivi nutrizionali, di fitness, peso e idratazione.

FitTracker non è la solita applicazione per diete restrittive: è un'applicazione che aiuterà l'utente a conoscere le sue abitudini, a capire cosa mangia, a fare scelte alimentari consapevoli più sane e a trovare, persino, sostegno e motivazione per raggiungere gli obiettivi di benessere prefissati. Utilizzare FitTracker è come avere un dietologo, un personal trainer e un mental coach sempre a portata di mano.

L'applicazione permette, inoltre, di personalizzare la dieta con la possibilità di aggiungere piatti preferiti e creare ricette personali, rendendo piacevole e divertente il controllo della propria alimentazione.

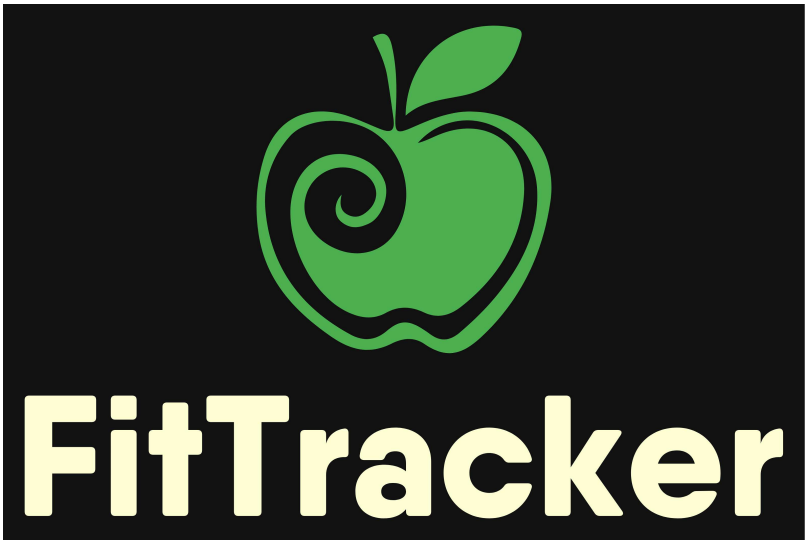


Figura 1: Logo dell'applicazione

2 Sintesi dell'approccio

L'approccio utilizzato per lo sviluppo dell'applicazione è stato quello di iniziare dall'analisi di altre applicazioni simili a pagamento per valutarne le funzionalità e individuarne i limiti.

Si è proceduto, quindi, alla stesura di una lista di funzionalità ritenute importanti e che l'applicazione avesse dovuto svolgere, con l'idea che l'interfaccia dovesse essere la più user-friendly possibile.

Sono state poi realizzate due applicazioni: una nativa per la piattaforma Android, sviluppata in Kotlin [1], che risulta quindi compatibile solo con smartphone che hanno questo sistema operativo installato, e una Flutter, in questo caso l'applicazione è disponibile sia per cellulari Android che iOS.

L'applicazione con le funzionalità più complesse ed interessanti è, tuttavia, quella sviluppata solo per Android [1], di conseguenza viene riportata soltanto l'implementazione riguardante quest'ultima.

3 App in Android (Kotlin)

3.1 Requisiti

I requisiti funzionali e non funzionali sono delle azioni/regole che la nostra applicazione dovrà rispettare.

Requisiti funzionali

REQUISITO	DESCRIZIONE
RF1 Login	L'applicazione dovrà gestire l'accesso
RF2 Logout	L'applicazione dovrà gestire la disconnessione
RF3 Registrazione	L'applicazione dovrà gestire la registrazione dell'utente
RF3 Modifica profilo	L'applicazione dovrà gestire la modifica dei dati personali dell'utente
RF4 Recupero password	L'applicazione dovrà gestire il recupero della password
RF5 Registrazione dei datidell'utente	L'applicazione dovrà gestire il salvataggio dei dati dell'utente
RF6 Salvataggio acqua	L'applicazione dovrà gestire il conteggio dell'acqua bevuta giornalmente
RF7 Salvataggio calorie	L'applicazione dovrà gestire il conteggio delle calorie quotidiane
RF8 Salvataggio carboidrati	L'applicazione dovrà gestire il conteggio dei grammi di carboidrati
RF9 Salvataggio proteine	L'applicazione dovrà gestire il conteggio dei grammi di proteine
RF10 Salvataggio grassi	L'applicazione dovrà gestire il conteggio dei grammi di grassi
RF11 Ricerca alimenti	L'applicazione dovrà gestire la ricerca degli alimenti da salvare nel diario
RF12 Ricerca esercizi	L'applicazione dovrà gestire la ricerca degli esercizi da salvare nel diario
RF13 Ricerca barcode	L'applicazione dovrà gestire la ricerca dell'alimento tramite barcode
RF14 Salvataggio alimenti personali	L'applicazione dovrà gestire il salvataggio degli alimenti/pasti/ricette personali dell'utente
RF15 Salvataggio alimenti preferiti	L'applicazione dovrà gestire il salvataggio degli alimenti preferiti dell'utente
RF16 Salvataggio esercizi personali	L'applicazione dovrà gestire il salvataggio degli esercizi personali dell'utente

REQUISITO	DESCRIZIONE
RF17 Salvataggio esercizi preferiti	L'applicazione dovrà gestire il salvataggio degli esercizi preferiti dell'utente
RF18 Aggiunta al diario	L'applicazione dovrà gestire il salvataggio nel diario delle calorie assunte e bruciate
RF19 Modifica alimenti personali	L'applicazione dovrà gestire la modifica degli alimenti personali
RF20 Modifica esercizi personali	L'applicazione dovrà gestire la modifica degli esercizi personali
RF21 Rimozione degli alimenti selezionati	L'applicazione dovrà gestire la rimozione degli alimenti selezionati nel diario
RF22 Rimozione degli esercizi selezionati	L'applicazione dovrà gestire la rimozione degli esercizi selezionati nel diario
RF23 Rimozione degli esercizi preferiti	L'applicazione dovrà gestire la rimozione degli esercizi preferiti
RF24 Rimozione degli alimenti preferiti	L'applicazione dovrà gestire la rimozione degli alimenti preferiti
RF25 Statistiche filtrate	L'applicazione dovrà gestire le statistiche dei consumi dell'utente secondo degli intervalli di tempo
RF26 Selezionare dieta	L'applicazione dovrà gestire la selezione della dieta dell'utente e modificare i macro-nutrienti in base ad essa
RF27 Funzioni	L'applicazione dovrà gestire le funzioni utili all'utente all'interno dell'applicazione
RF28 Calcolo del fabbisogno	L'applicazione dovrà gestire e calcolare il fabbisogno calorico giornaliero dell'utente in base ai suoi dati.

Requisiti non funzionali

REQUISITO	DESCRIZIONE
RNF1 Kotlin	L'applicazione dovrà essere scritta completamente in Kotlin
RNF2 Intuitiva	L'applicazione dovrà essere la più intuitiva e semplice possibile
RNF3 Fluida	L'applicazione dovrà compiere azioni e dovrà essere possibile navigare al suo interno in maniera fluida e senza crash
RNF4 Estetica	L'applicazione dovrà essere bella esteticamente
RNF4 UserFriendly	L'applicazione dovrà essere facilmente usabile
RNF5 Privacy	L'applicazione dovrà rispettare la privacy dell'utente secondo le norme dell'UE
RNF6 Permessi fotocamera	L'applicazione richiede l'accesso alla fotocamera
RNF7 Connessione internet	L'utente deve avere una connessione ad internet disponibile

3.2 Utilizzo in Android

Di seguito è riportato lo schema dei casi d'uso che riassume le principali funzionalità dell'applicazione.

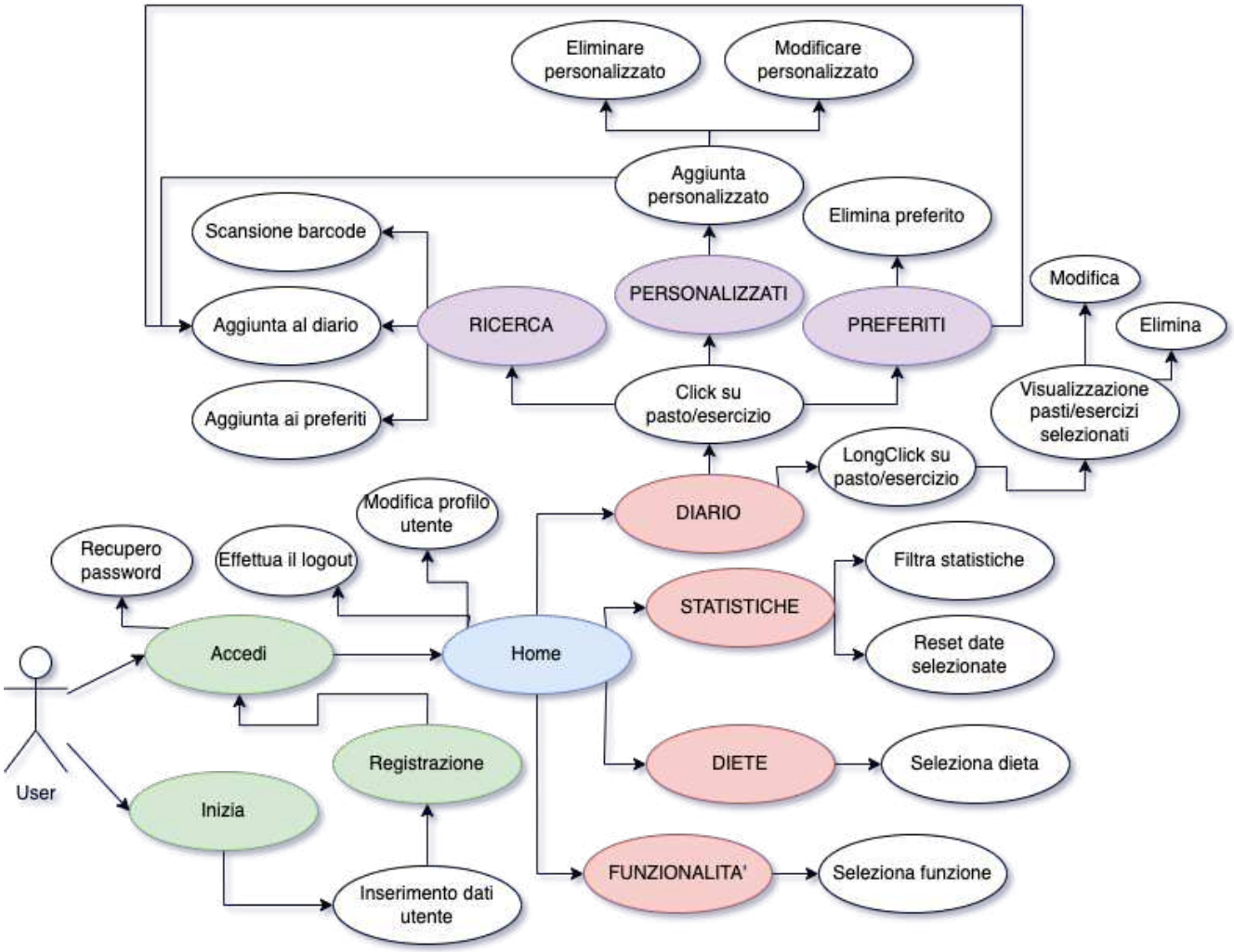


Figura 2: Casi d'uso dell'applicazione

Accesso e Registrazione

Come si può notare, all'inizio l'utente può svolgere due azioni: effettuare l'accesso cliccando sul pulsante "Accedi" oppure effettuare la registrazione cliccando il pulsante "Inizia"

Effettuando una nuova registrazione, appariranno una serie di schermate dove l'utente potrà inserire i dati personali utili a calcolare il proprio fabbisogno calorico giornaliero. Infine inserirà il proprio indirizzo email ed una password che gli serviranno per l'accesso e il mantenimento dei dati sul cloud.

Quando invece l'utente è già registrato, può accedere all'applicazione cliccando il pulsante "Accedi". Inserendo il proprio indirizzo email e la propria password avrà accesso alla schermata home. Nel caso in cui l'utente abbia dimenticato la password è possibile recuperarla tramite l'invio di un link di reset utilizzando l'email indicata.

Schermata Home

Una volta effettuato l'accesso, l'utente si troverà nella schermata principale dell'applicazione: la Home. La Home è suddivisa in quattro sezioni: Diario, Statistiche, Diete e Funzionalità. Sempre nella schermata di Home, è presente una toolbar, con un menù a tendina, contenente due voci: Profilo e Logout.
Cliccando sul Profilo, verrà aperta una schermata con tutti i dati dell'utente, i quali possono essere modificati per cambiare il valore del fabbisogno calorico giornaliero. Cliccando su Logout, invece, si effettuerà la disconnessione dall'account.

Schermata Home: Diario

Appena aperta l'applicazione, l'utente si troverà nella sezione Diario, nella quale potrà visualizzare il suo fabbisogno calorico giornaliero, con le calorie rimanenti, assunte e bruciate. Inoltre troverà anche delle barre di progresso le quali mostrano i grammi di macro nutrienti assunti e quelli da assumere ancora. Questi dati sono stati generati sulla base dei dati dell'utente inseriti nella registrazione e in base alla dieta che l'utente ha selezionato. Di base l'utente partirà da una dieta Mediterranea.

Sempre nella sezione del Diario, l'utente può andare a registrare e visualizzare i pasti (e l'attività fisica) consumati nel corso della giornata ed i bicchieri d'acqua bevuti (ogni bicchiere conta 250 ml di acqua, per un totale di 2 litri d'acqua al giorno). Cliccando sul pulsante dei pasti o sul pulsante esercizio, si aprirà una nuova scheda composta da tre schermate: Ricerca, Personalizzati e Preferiti, mentre la lunga pressione di questi pulsanti aprirà una schermata pop-up per mostrare i prodotti/esercizi già salvati. E' importante specificare che per avere una maggiore pulizia nella lista dei pasti/esercizi già selezionati, l'utente, qualora aggiungesse un pasto/esercizio già selezionato precedentemente, esso verrà sovrascritto. Per questo abbiamo dato la possibilità all'utente di andare a modificare, oltre che ad eliminare, i pasti/esercizi che sono già stati salvati nel diario.

Ricerca

Nella sezione Ricerca è possibile andare a cercare i pasti/esercizi tramite barra di ricerca, oppure scansionando il codice a barre del prodotto cliccando sul pulsante in alto a destra. Nel caso in cui si ricerchi il prodotto/esercizio per nome all'interno della barra di ricerca, cliccandolo si aprirà una schermata con tutti i valori nutrizionali dello stesso. Tra questi troviamo le calorie, i grammi di carboidrati, i grammi di grassi e i grammi di proteine. Le informazioni nutrizionali sono calcolate su 100 gr. di prodotto. Inoltre possiamo trovare delle informazioni aggiuntive per quanto riguarda il prodotto, come il brand, la categoria a cui appartiene e la descrizione. In fondo alla schermata è possibile inserire la quantità di prodotto da aggiungere.
Questi alimenti sono presi da un database online, quindi potrebbe essere possibile che non ci siano tutti i prodotti, oppure che non abbiano le informazioni aggiuntive o immagini illustrative. Per selezionare una quantità di grammi non multipla di 100, basta mettere la quantità come numero decimale.
Una volta selezionata la quantità, è possibile aggiungere il prodotto al diario, oppure salvarlo tra i preferiti.
Per quanto riguarda la ricerca degli esercizi, essa dovrà essere fatta in inglese.

Personalizzati

In questa sezione è possibile aggiungere un pasto rapido personalizzato/ricetta/alimento. Infat-

ti, cliccando sul pulsante "+" in alto, si aprirà una schermata contenente i campi da compilare con le informazioni relative ad esso. Ciò consente all'utente di avere un database infinito di prodotti, infatti verranno salvati all'interno del cloud, permettendo all'utente di poter cambiare tranquillamente dispositivo, senza perdere i propri prodotti.
Una volta aggiunto il prodotto personalizzato, esso verrà visualizzato nella lista all'interno di personalizzati. Cliccandoci sopra sarà poi possibile compiere diverse azioni, tra cui: modificare le informazioni relative, aggiungerlo al diario (la quantità verrà scelta dall'utente e non è più basato su 100 gr. di prodotto) ed eliminarlo dalla lista.

Preferiti

All'interno di questa sezione vengono visualizzati i prodotti salvati come preferiti. Anch'essi vengono salvati sul cloud. Selezionando un prodotto preferito si potranno compiere, anche qui, diverse azioni, tra cui: aggiungere il pasto al diario oppure rimuoverlo dalla lista dei preferiti.

Schermata Home: Statistiche

La seconda sezione che troviamo nella Home sono le Statistiche. All'interno di questa schermata, l'utente può verificare diverse categorie di statistiche, come ad esempio le calorie consumate, i litri d'acqua bevuti, i grammi di carboidrati, proteine e grassi consumati ed i giorni in cui l'utente ha raggiunto l'obiettivo di 2 litri di acqua bevuti.
Si possono effettuare due tipi di filtraggio: inserendo un intervallo di date, ovvero una data di inizio e una data di fine, oppure, lasciando vuoti tali spazi, è possibile avere un resoconto totale da quando l'utente si è registrato.

Schermata Home: Dieta

Nella sezione Dieta, l'utente può selezionare una qualsiasi dieta secondo le sue esigenze. Cliccando su una di esse è possibile andare a vedere nel dettaglio cosa propone. Infatti ognuna di esse ha diverse percentuali di macro nutrienti, le quali verranno utilizzate per ricalcolare i grammi di quest'ultimi in base al fabbisogno calorico dell'utente.
Tra le diete che abbiamo inserito troviamo: la dieta "Climatica" (50% carb. , 20% proteine, 30 % grassi), la dieta "Keto leggera" (19% carb. , 15% proteine, 66 % grassi), la "Keto media" (9% carb. , 15% proteine, 76% grassi), la dieta "Mangiare pulito" (40% carb. , 25% proteine, 35 % grassi), la "Mediterranea" (40% carb. , 20% proteine, 40 % grassi) e la dieta "Scandinava" (40% carb. , 30% proteine, 30 % grassi). La dieta attualmente selezionata sarà segnata da un contorno rosso attorno alla card della stessa.

Schermata Home: Funzioni

Questa è una sezione dedicata a delle funzioni inerenti al fitness e al benessere fisico dell'utente. All'interno del **Capitolo 4** di questo documento verranno esposte le procedure di implementazione di ciascuna funzionalità che troviamo all'interno dell'applicazione.

3.3 Architettura in Android

Per lo sviluppo di questa versione dell'applicazione, abbiamo utilizzato Kotlin come linguaggio, e abbiamo utilizzato Android Studio come IDE.

Per quanto riguarda l'aspetto tecnico dell'applicazione, abbiamo utilizzato l'architettura MVVM. Non abbiamo implementato nessun database locale, infatti abbiamo optato per l'utilizzo di un database remoto, utilizzando Firebase. Firebase è usato sia per l'autenticazione che per lo storage dei dati persistenti dell'utente (FirebaseAuth e FirebaseStore).

Per la ricerca dei prodotti e degli esercizi abbiamo utilizzato due API diverse, rispettivamente Edamam e Ninjas.

Database remoto

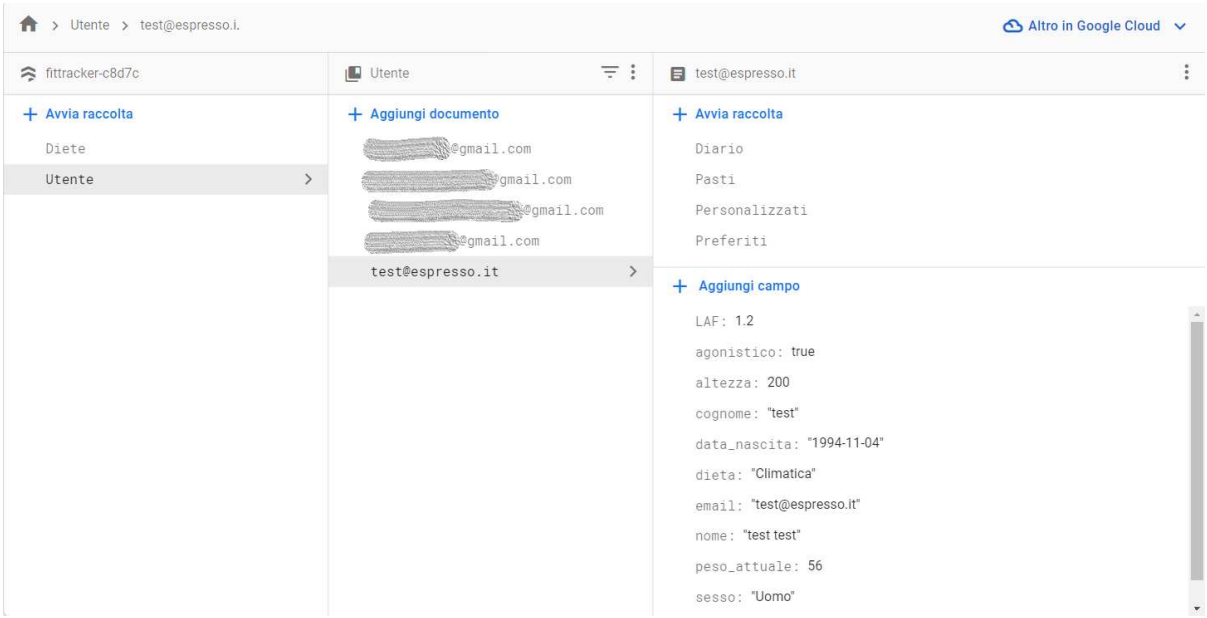


Figura 3: Questo è uno screen che mostra la struttura del database di Firebase (Firestore), ci troviamo nella collection "Utente" in cui troviamo tutti i nostri utenti suddivisi per email. La struttura del database è costituito da un alternarsi di collection e document, l'ultimo document potremo popolarlo con degli attributi (campi chiave-valore). Nel nostro esempio ci troviamo sull'utente "test@espresso.it" e notiamo come al suo interno abbiamo sia i dati personali dell'utente, sia altre raccolte, nelle quali sono salvati altri dati inerenti alla collection selezionata.

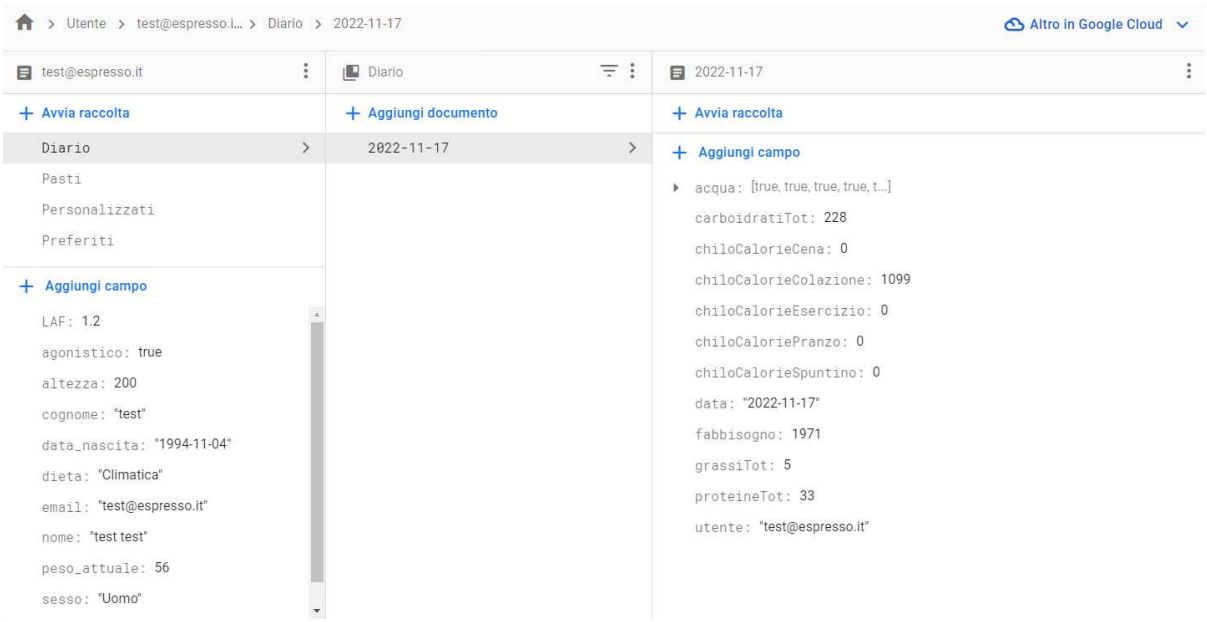


Figura 4: Questo screen mostra ciò che contiene una raccolta all'interno dell'utente selezionato. In questo caso ci troviamo nella collection "Diario", al suo interno i documenti sono suddivisi per data (che sarebbe l'id del document), al suo interno, a sua volta troviamo i campi chiave-valore che il diario deve contenere (bicchieri d'acqua, fabbisogno giornaliero, carboidrati, proteine, grassi, ecc.). In questo modo la ricerca dei diari avverrà per data, e questo ci aiuta nel filtraggio delle statistiche. Anche la collection "Pasti" ha la stesso logica di suddivisione dei document.

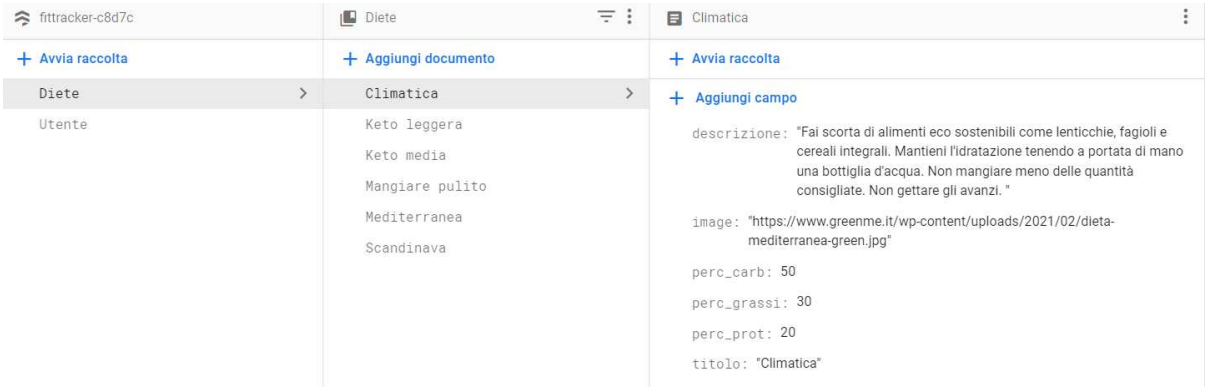


Figura 5: Struttura della collection delle diete. Avendo messo le diete nel database remoto, possiamo andare ad aggiornare, aggiungere e rimuovere le diete all'interno dell'applicazione senza andare a modificare il codice, in quanto le diete vengono aggiunte come item ad una RecyclerView attraverso la chiamata del ViewModel al DB.

3.4 Schermate in Android

Mockup FitTracker v1.0

Qui di seguito sono riportati i mockup della vecchia versione dell'applicazione. Per comodità sono stati divisi in diverse parti.
Le schermate superflue come ad esempio per la creazione di pasti personalizzati, ricette, messaggi e schermate ripetute o di importanza poco rilevante, sono state omesse per praticità.
Essendo dei mockup alcune schermate non saranno identiche al 100%, ma servono solamente come linee guida per la progettazione delle schermate definitive.

Login e Registrazione

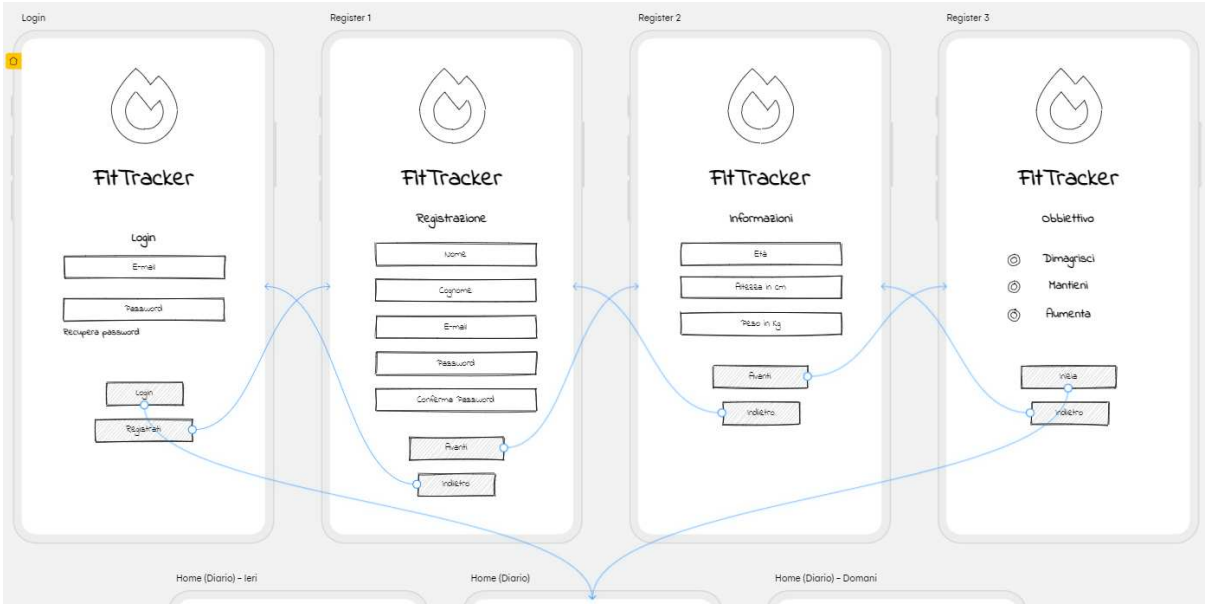


Figura 6: Dallo screen **Login** cliccando il pulsante "Login" accediamo alla **Home (Diario)**, mentre cliccando il pulsante "Registrati" verrà aperto lo screen **Register 1**, da cui poi partirà tutta la fase di registrazione. In queste schermate verranno aggiunte tutte le informazioni dell'utente. Inserite quest'ultime si verrà reindirizzati allo screen **Home (Diario)**.

Home (Diario)

La schermata **Home (Diario)** è la nostra schermata principale. Nella schermata stessa è possibile andare ad aggiungere i pasti della giornata, gli esercizi svolti e l'acqua assunta. Mentre in cima è presente un resoconto delle calorie rimanenti da assumere, quelle assunte e quelle bruciate in allenamento. E' possibile andare a visualizzare le schede relative alle giornate precedenti ed a quelle future grazie alle frecce di navigazione.

In fondo alla schermata abbiamo un **Bottom Navigation**, il quale ci permette di navigare all'interno dele diverse schermate dell'applicazione.

Bottom Navigation

Scegliendo uno degli elementi della Bottom Navigation possiamo accedere a sezioni diverse dell'applicazioni, ognuna con una diversa funzionalità. Da esse poi si potrà accedere ancora ad altre sotto-interfacce.

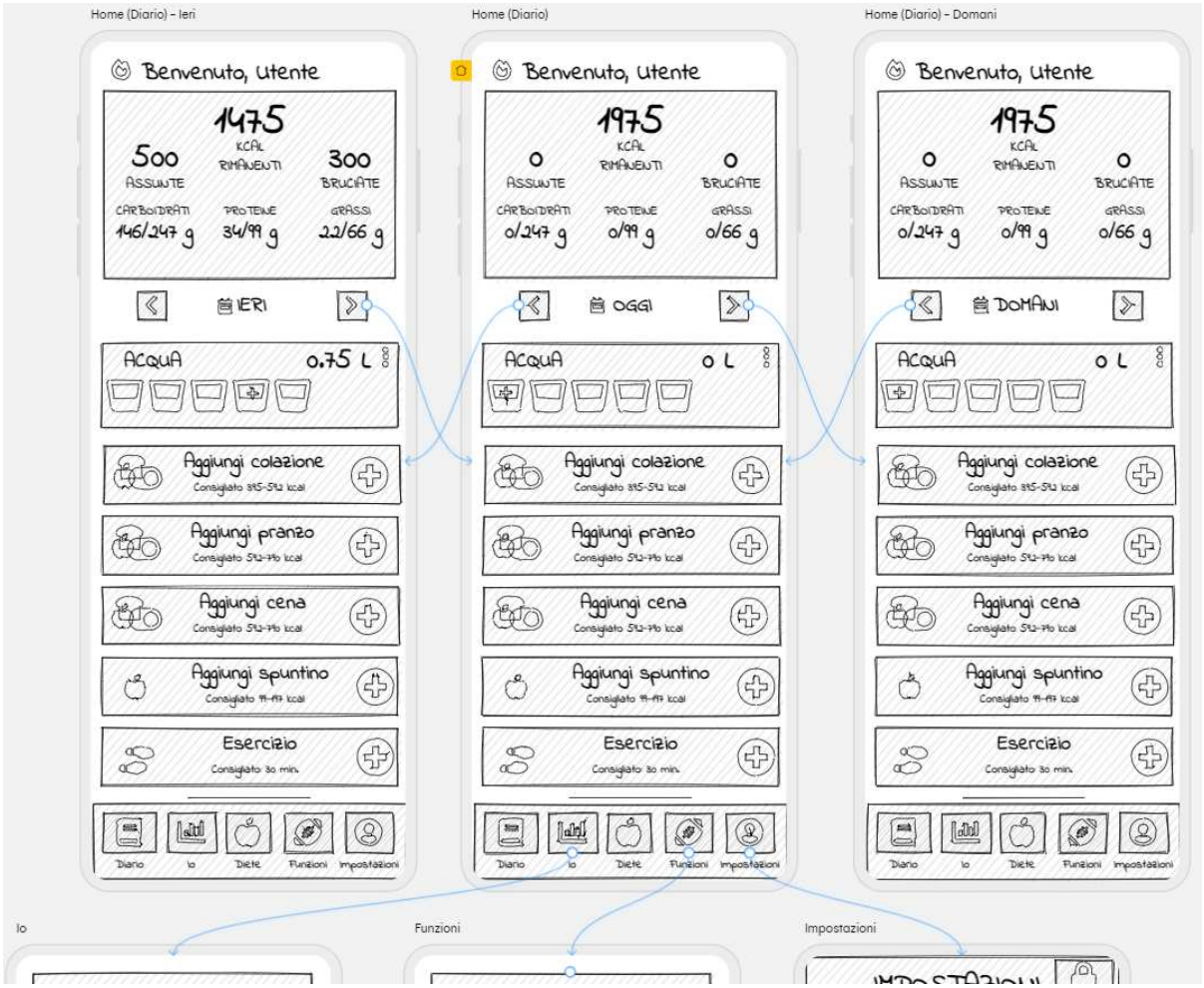


Figura 7: Cliccando sul "+" accanto ad ogni card è possibile aggiungere un alimento al pasto. Lo stesso vale con i bicchieri d'acqua. Aggiungendo degli esercizi, le calorie bruciate saranno tenute in considerazione nel resoconto delle calorie rimanenti.

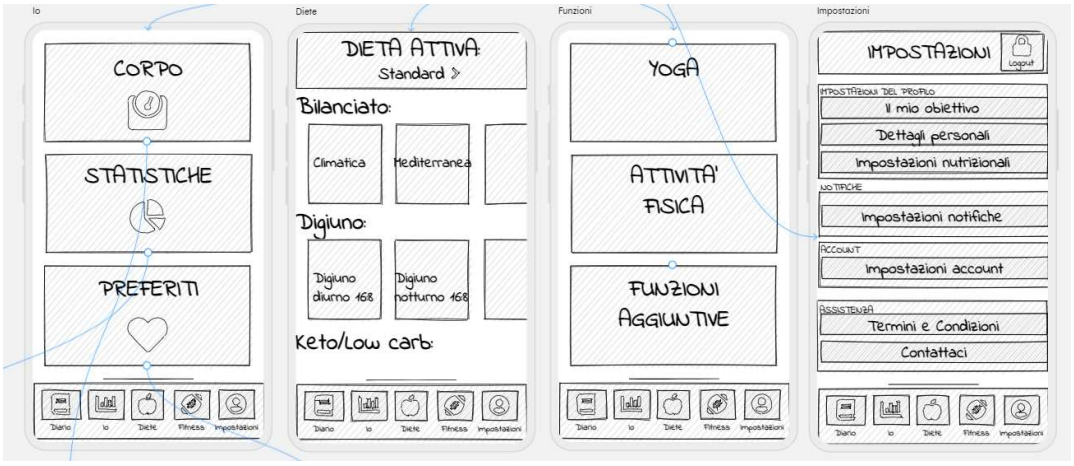


Figura 8: Le quattro schermate con la visualizzazione delle sotto viste

Sezione "IO"

Prendiamo come esempio la sezione "IO", la quale si divide in "Corpo", "Statistiche" e "Preferiti". Ognuna di queste card porta ad una interfaccia diversa, con una sua funzionalità specifica.

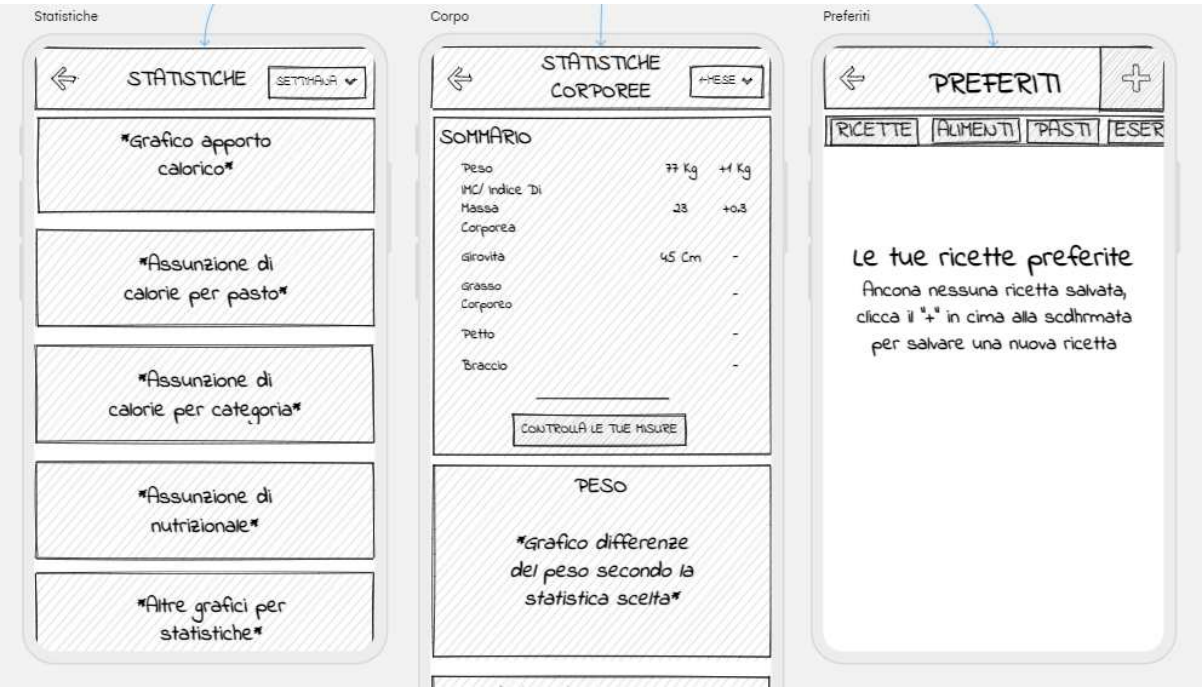


Figura 9: Sezione Io

Aggiunta di un alimento ad un pasto

Cliccando il "+" accanto ad un pasto si potrà avviare la ricerca dell'alimento tramite la barra di ricerca oppure scannerizzando il codice a barre del prodotto. E' possibile anche selezionare degli alimenti salvati tra i preferiti.

Lo stesso ragionamento lo facciamo per gli esercizi.

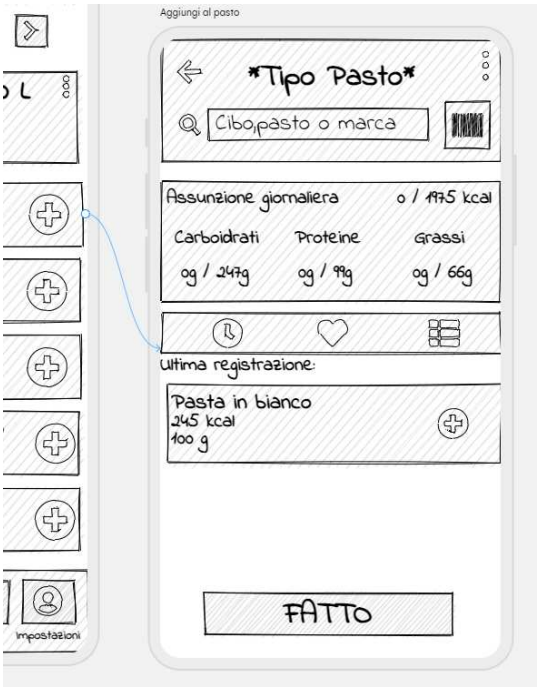


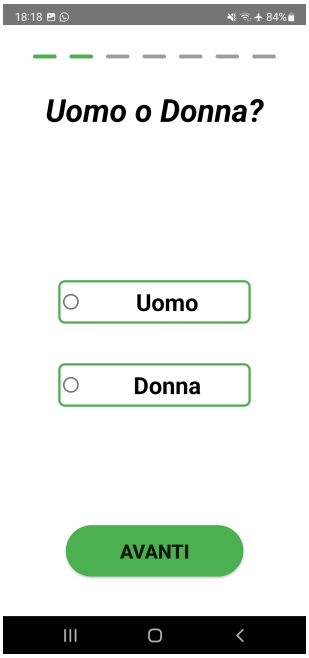
Figura 10: Sezione per aggiungere un pasto

Mockup FitTracker v2.0

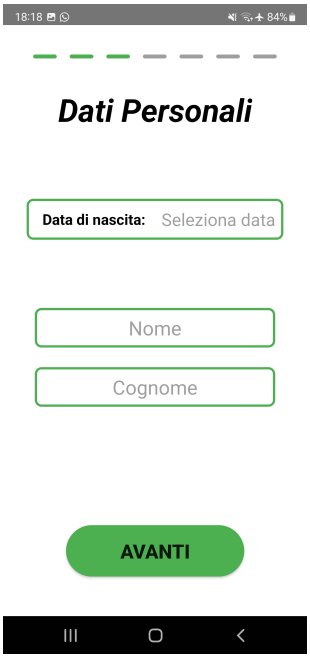
Di seguito mostriamo le schermate definitive della nostra applicazione sviluppata per dispositivi Android. Vengono mostrate tutte le Activity ed i Fragments in cui l'utente può navigare e con cui può interagire.

(1) InizioActivity

(2) StileDiVitaFragment



(3) SessoFragment



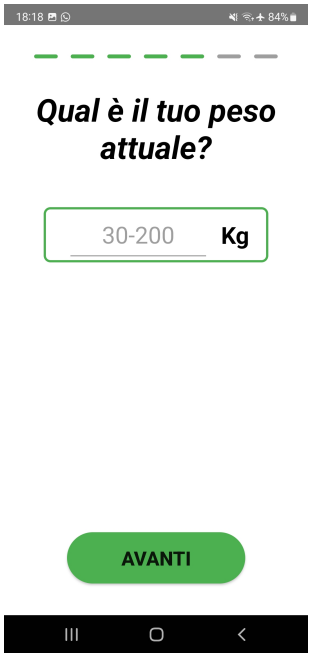
(4) DatiPersonaliFragment

La prima schermata con cui si va ad interagire è la InizioActivity, da qui possiamo prendere due strade: la prima ci porta allo StileDiVitaFragment, la seconda al LoginActivity, usato per l'accesso.
Con lo StileDiVitaFragment si inizia il percorso di registrazione, in cui l'utente va a specificare tutti i suoi dati personali utili per la creazione del diario. Infatti il calcolo delle calorie gior-

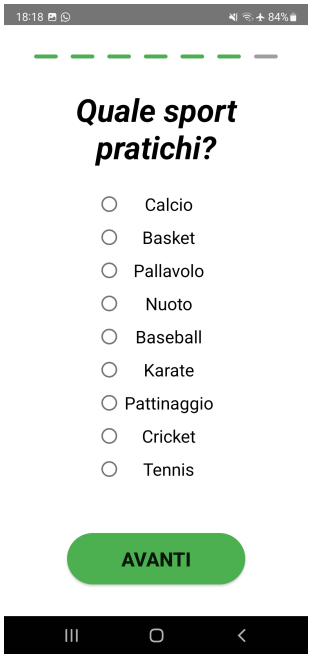
naliere dipende, oltre dall'altezza, il peso e il sesso, dal livello di attività fisica che si svolge settimanalmente.

Ovviamente i campi sono tutti obbligatori. La data è controllata e l'età minima per registrarsi è di 15 anni. L'altezza inserita e il peso devono essere compresi rispettivamente tra i 130-300 cm e 30-200 Kg. Qualora uno degli input fosse sbagliato, sarà impossibile andare avanti.

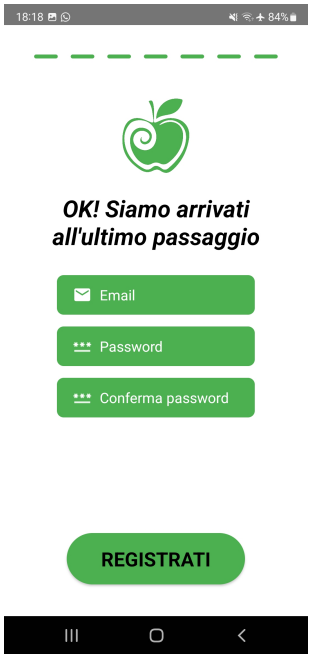
(5) AltezzaActivity



(6) PesoActivity



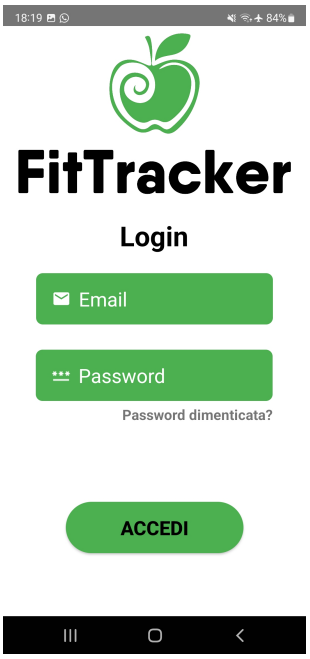
(7) SportActivity



(8) RegisterActivity

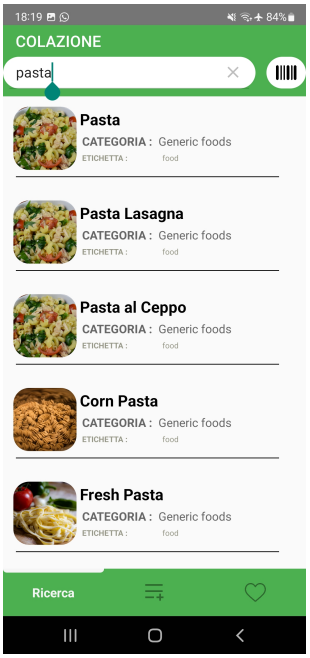
La schermata SportActivity è opzionale, in quanto viene aggiunta solamente se l'utente specifica di essere un atleta agonista in StileDiVitaActivity. La scelta dello sport non comporta alcuna modifica nel calcolo delle calorie, ma serve qualora di volesse fare una statistica sugli sport più praticati dagli utenti. Una volta completate tutte le informazioni, l'utente può registrarsi, inserendo la propria email e password.

Avvenuta la corretta registrazione dell'utente, si potrà effettuare il login all'interno della LoginActivity. Inoltre, all'interno di essa, sarà possibile andare a recuperare la password qualora si fosse dimenticata. Il recupero password avviene tramite link inviato all'email specificata.
Effettuato l'accesso, viene aperta la HomeActivity, nella sezione DiarioFragment, il quale contiene tutte le informazioni relative ai pasti (calorie, macro nutrienti), esercizi e acqua del giorno corrente.

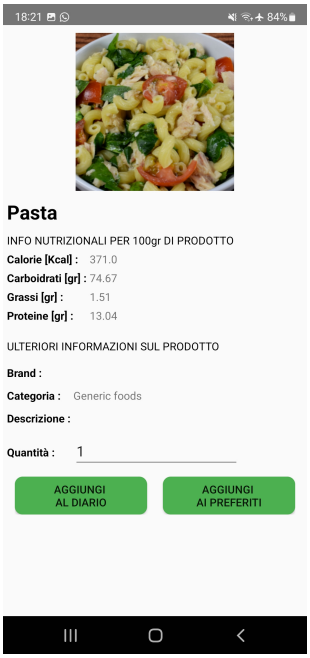


(9) LoginActivity

(10) DiarioFragment



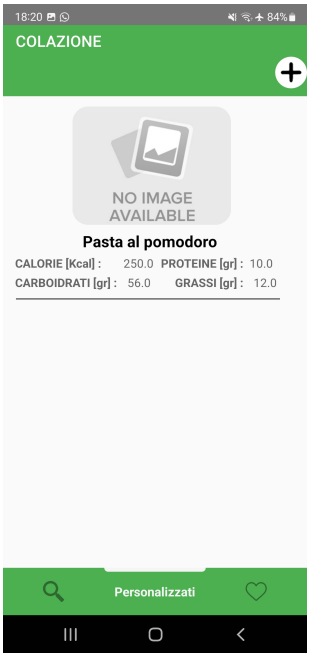
(11) RicercaFragment



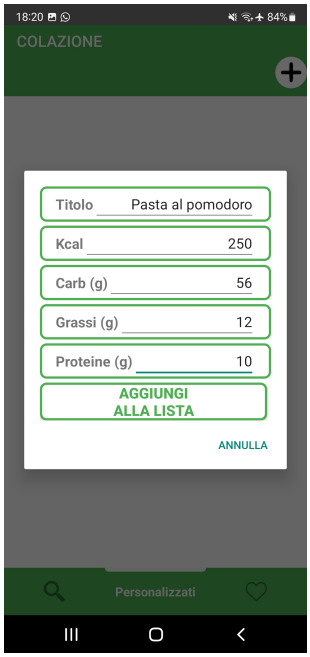
(12) Selezione item ricerca

Cliccando su uno dei pulsanti relativi ai pasti o agli esercizi, è possibile andare a ricercare (RicercaFragment) i pasti/esercizi da aggiungere al diario. Inoltre è possibile aggiungere propri pasti/esercizi alla lista personalizzati (PersonalizzatiFragment), oppure salvare dei prodotti/esercizi nella lista dei preferiti (PreferitiFragment).

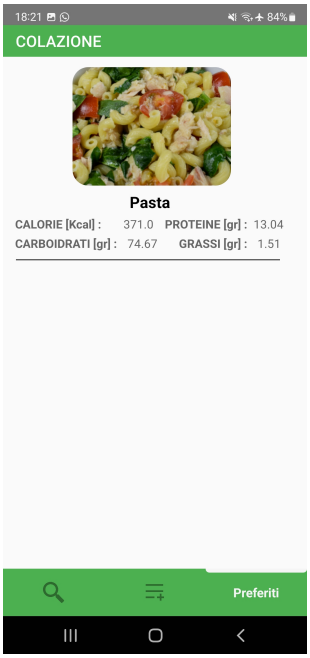
Cliccando su un elemento della lista, a seconda di dove ci si trova, si aprirà una finestra a pop-up per compiere diverse azioni con quell'item. Ad esempio cliccando un elemento nella lista dei Personalizzati è possibile andare a modificare il pasto personalizzato, aggiungerlo al diario, oppure eliminarlo. Lo stesso vale per gli elementi nella lista Preferiti.



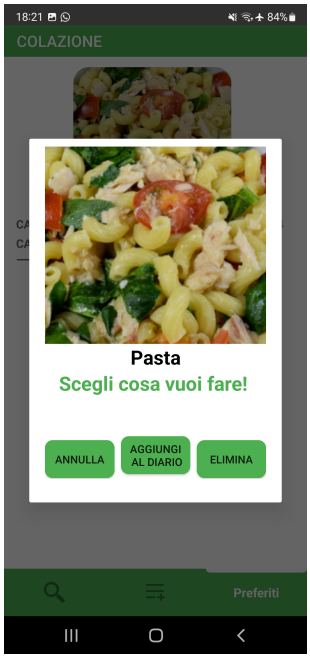
(13) PersonalizzazioneFragment



(14) Aggiunta item personale



(15) PreferitiFragment



(16) Selezione item preferito

Mentre, se invece di cliccare sul pulsante relativo al pasto/esercizio si effettua una pressione prolungata (long click) si aprirà la lista degli elementi selezionati, che è possibile modificare o rimuovere dal diario, ripristinando così le calorie assunte.

Le altre tre schermate possibili della HomeActivity sono: StatisticheFragment, DieteFragment e FunzioniFragment. La schermata delle statistiche permette all'utente di controllare diversi indici in un intervallo di tempo fissato oppure in totale.
La sezione diete permette all'utente di cambiare la dieta, infatti ogni dieta ha delle diverse percentuali di macro nutrienti, di conseguenza cambieranno i grammi giornalieri di quest'ultimi. La dieta selezionata è evidenziata da una cornice rossa intorno alla card della stessa. Cliccando sopra una di esse si potrà andare a vedere nel dettaglio cosa consiglia la dieta.

(17) StatisticheFragment

(18) DieteFragment

(19) FunzioniFragment

FunzioniFragment è riservato a delle funzioni che verranno implementate nelle versioni future. Esso conterrà degli strumenti utili all'utente sempre nell'ambito del fitness e del benessere psico-fisico.

3.5 Sviluppo

Classi

Package: aggiungi_esercizio
All'interno di questo package troviamo tutte le classi che riguardano la parte di ricerca degli esercizi, salvataggio di esercizi personali e preferiti, e la visualizzazione delle informazioni sulle calorie bruciate.

AggiungiEsercizioActivity: questa è l'activity che mostra la ricerca degli esercizi, la lista degli esercizi personali e la lista di quelli preferiti. E' composta da una BottomNavigation che ci permette di selezionare le schemate di RicercaEserciziFragment, PersonalizzatiEsercizioFragment e PreferitiEserciziFragment. Di conseguenza va a gestire la selezione corretta della navigazione.

AggiungEserciziiViewModel: la classe ViewModel dell'AggiungiEsercizioActivity, si occupa del collegamento del DB con le viste. Tramite i metodi ottiene dal DB gli esercizi personalizzati e preferiti e li salva in una lista di tipo Esercizio. Inoltre recupera anche dal DB dell'API l'esercizio selezionato nella ricerca e impostata le calorie bruciate nella schemata del diario.

MyAdapterPrefPersEsercizio e **MyAdapterRicercaEsercizio:** sono gli adapter che si occupano degli item delle RecyclerView all'interno dei fragment dell'AggiungiEsercizioActivity.

PersonalizzatiEsercizioFragment, PreferitiEserciziFragment e **RicercaEserciziFragment:** sono i fragment collegati alla bottom navigation dell'AggiungiEsercizioActivity. All'interno di essi gestiamo le azioni da compiere relativamente agli item cliccati e dell'impostazione dei layout. Inoltre nel RicercaEserciziFragment si gestisce anche il passaggio degli attributi da un'activity all'altra.

Package: aggiungi_pasto
All'interno di questo package troviamo tutte le classi che riguardano la parte di ricerca dei prodotti, salvataggio di pasti personali e preferiti, e la visualizzazione delle informazioni nutrizionali.

AggiungiActivity: questa è l'activity che mostra la ricerca dei pasti, la lista dei pasti personali e la lista dei preferiti. E' composta da una BottomNavigation che ci permette di selezionare le schemate di RicercaFragment, PersonalizzatiFragment e PreferitiFragment. Di conseguenza va a gestire la selezione corretta della navigazione.

AggiungiViewModel: la classe ViewModel dell'AggiungiActivity, si occupa del collegamento del DB con le viste. Tramite i metodi ottiene dal DB i pasti personalizzati e preferiti e li salva in una lista di tipo Pasto. Inoltre recupera anche dal DB dell'API il prodotto selezionato nella ricerca.

MyAdapterPrefPers e **MyAdapterRicerca:** sono gli adapter che si occupano delle RecyclerView all'interno dei fragment dell'AggiungiActivity.

PersonalizzatiFragment,PreferitiFragment e **RicercaFragment:** sono i fragment collegati alla bottom navigation dell'AggiungiActivity. All'interno di essi gestiamo le azioni da compiere relativi agli item cliccati o all'impostazione dei layout. Inoltre nel RicercaFragment si gestisce anche il passaggio degli attributi da un'activity all'altra.

Package: autenticazione
All'interno di questo package troviamo tutte le classi che riguardano la registrazione dei dati personali dell'utente, gestione dell'autenticazione e il recupero della password.

InizioActivity: questa è l'activity da dove tutto parte, a seconda del pulsante che si clicca, l'utente può registrarsi oppure effettuare l'accesso.

ConosciamociActivity: è l'activity che parte qualora l'utente scelga di registrarsi, partiranno una serie di fragments nei quali si potrà andare ad inserire tutti i dati personali.

RegisterActivity: è l'activity finale dopo che l'utente ha completato con successo il salvataggio dei dati. Questa activity gestisce la registrazione dell'utente e vede se l'email è già in uso oppure no.

LoginActivity: nel caso in cui l'utente sia già registrato, gli basterà effettuare l'accesso e questa è l'activity che gestisce tutto.

RecuperoActivity: è l'activity che gestisce il recupero della password dell'utente qualora se la fosse dimenticata.

AltezzaFragment, DatiPersonaliFragment, PesoAttualeFragment, SessoFragment, SportFragment, StileVitaFragment: sono i fragments che gestiscono l'input dei dati dell'utente e se li passano da fragment a fragment fino alla RegisterActivity, la quale si occupa di andarli a salvare definitivamente nel FirebaseDB.

Package: databaseFB
All'interno di questo package troviamo tutte le classi che riguardano il collegamento al FirestoreDB e il salvataggio degli esercizi, pasti, diari, personalizzazioni e preferiti per ogni utente, organizzati per date.

FirebaseDB: è una classe che gestisce l'istanza al FirebaseFirestore e si occupa anche di prendere l'email dell'utente attualmente autenticato, da passare poi ai path delle collection e dei document del db.

DiarioDB: è la classe che gestisce il settaggio del diario nel db (passandogli la data corrente) e l'acquisizione degli stessi dall'interno del database.

DietaDB: gestisce la restituzione delle diete dal database.

EsercizioDB: è la classe che gestisce il settaggio dell'esercizio nel db (passandogli la data corrente) e l'acquisizione degli stessi dal database.

PersonalizzatiDB: è la classe che gestisce il settaggio dei personalizzati nel db (passandogli il pasto) e l'acquisizione degli stessi dal database.

PreferitiDB: è la classe che gestisce il settaggio dei preferiti nel db (passandogli il pasto) e la restituzione degli stessi dal database.

ProdottoDB: è la classe che gestisce il settaggio del prodotto nel db (passandogli la data) e la restituzione degli stessi dal database.

UtenteDB è la classe che gestisce il settaggio dei dati dell'utente nel db (passandogli l'email) e la restituzione degli stessi dal database.

Package: diario
All'interno di questo package troviamo tutte le classi che riguardano il fragment principale dell'applicazione, in cui vengono visualizzate le calorie rimanenti, quelle consumate e quelle bruciate. Vengono, inoltre, visualizzati i pasti e gli esercizi con le relative calorie e l'acqua bevuta.

DiarioFragment: è la classe che contiene la creazione della view e il settaggio del diario dell'utente.

DiarioViewModel: è la classe che collega il database e la vista, andando a prendere i dati dell'utente per calcolare il fabbisogno calorico e i macro nutrienti.

MyAdapterEserciziSel e **MyAdapterSelezionati:** sono gli adapter che vanno a gestire gli item nelle RecyclerView per andare a modificare ed eliminare i pasti e gli esercizi già selezionati e salvati nel diario.

```
1  //Metodo per impostare i macro nutrirenti in base alla dieta e al fabbisogno
       giornaliero
2  private fun setMacro(){
3          viewModelScope.launch {
4              val utente = utenteDB.getUtente(auth.currentUser?.email!!)
5              val dieta = dietaDB.getDieta(utente.dieta)
6              _carboidratiMax.value = ((diario.value!!.fabbisogno*(dieta.
       perc_carb.toDouble()/100.0)) / 4).toInt() //1gr di carbo = 4Kcal
7              _proteineMax.value = ((diario.value!!.fabbisogno*(dieta.
       perc_prot.toDouble()/100.0)) / 4).toInt() //1gr di prot = 4Kcal
8              _grassiMax.value = ((diario.value!!.fabbisogno*(dieta.perc_prot.
       toDouble()/100.0)) / 9).toInt()//1gr di grassi = 9Kcal
9              _diarioSettato.value = !(_diarioSettato.value)!!
10         }
11     }
```

```
1
2
3  //Metodo per calcolare il fabbisogno giornaliero in base ai dati dell'utente
4  private fun calculateFabbisogno(utente: Utente) : Double{
5          val today = LocalDate.now()
6          val birthday: LocalDate = LocalDate.parse(utente.data_nascita)
7          val period: Period = Period.between(birthday, today)
8          if(utente.sesso == "Uomo")
9              return ((66 + (13.7 * utente.peso_attuale) + (5 * utente.altezza
       ) - (6.8 * period.years)) * utente.LAF)
10         else
11             return ((65 + (9.6 * utente.peso_attuale) + (1.8 * utente.
       altezza) - (4.7 * period.years)) * utente.LAF)
12     }
```

Package: diete
All'interno di questo package troviamo tutte le classi che riguardano il fragment delle diete che l'utente può selezionare.

DieteFragment: è la view che contiene la RecyclerView contenente le diete a griglia.

DieteViewModel: è la classe che gestisce la selezione della dieta dal database e il cambio di dieta dell'utente.

MyAdapterDiete: è l'adapter che gestisce la RecyclerView e la disposizione delle diete al suo interno.

Package: home
All'interno di questo package troviamo tutte le classi che riguardano la Home, la schermata che racchiude tutto.

HomeActivity: è l'activity che gestisce la navigazione all'interno dell'applicazione e tra le schermate principali. Inoltre gestisce la toolbar per andare a cambiare i dati dell'utente e il logout.

HomeViewModel: è la classe che gestisce la disconnessione dell'account.

Package: model
All'interno di questo package troviamo tutte le data class usate per i model degli oggetti Diario, Dieta, Pasto e Utente.

Package: pasto
All'interno di questo package troviamo tutte le classi usate per la gestione dei pasti già selezionati.

PastoActivity: è la classe che gestisce l'activity che mostra le informazioni del pasto selezionato dalla lista.

PastoViewModel: è la classe che gestisce le azioni da svolgere sul pasto che è stato già selezionato, per poi andare ad aggiornare le calorie e i macro nutrienti qualora si eliminasse o modificasse.

Package: prodotto
All'interno di questo package troviamo tutte le classi usate per la gestione dei pasti.

ProdottoActivity: è la classe che gestisce l'activity che mostra le informazioni del prodotto ricercato nel database. Poi da qui si potrà scegliere se metterlo nel diario oppure tra i preferiti.

ProdottoViewModel: è la classe che gestisce il salvataggio delle informazioni sul pasto qualora si salvasse nel diario oppure il salvataggio nella lista dei preferiti. Collega, quindi, la nostra app al database, contenente le informazioni relative alle calorie consumate in quella data su quel relativo pasto.

Package: profilo
All'interno di questo package troviamo tutte le classi usate per la gestione del profilo.

ProfiloActivity: è la classe che gestisce l'activity che mostra le informazioni dell'utente, prende le informazioni dal database grazie al ViewModel e imposta le diverse liste con i valori salvati.

ProfiloViewModel: è la classe che collega l'activity ProfiloActivity al database e si occupa nel restituire ed aggiornare i dati dell'utente, il cambio di password e ricalcolare il fabbisogno in base all'altezza, peso, sesso ed età qualora l'utente li modificasse.

Package: scanner
All'interno di questo package troviamo tutte le classi usate per la gestione del profilo.

ScannerActivity: è la classe che gestisce l'activity dell'apertura e dei permessi d'accesso della fotocamera del dispositivo e la ricezione di un codice a barre oppure un QR code.

Package: statistiche
All'interno di questo package troviamo tutte le classi usate per la gestione delle statistiche.

StatisticheFragment: è la classe che gestisce il fragment della schermata delle statistiche, l'impostazione dell'intervallo di tempo e le view che mostrano i risultati.

StatisticheViewModel: è la classe che gestisce la filtrazione dei dati nel database secondo l'intervallo inserito e li restituisce.

```
private fun ottieniStatistihe(data_inizio: String, data_fine: String)  {
        viewModelScope.launch {
            _statiticheLiveData.value = diarioDB.getStatistiche(data_inizio,
    data_fine)
            _kcalAssunte.value = 0
            _grCarboidrati.value = 0
            _grProteine.value = 0
            _grGrassi.value = 0
            _litriBevuti.value = 0.0
            _giorniAcqua.value = 0
            for (diario in _statiticheLiveData.value!!){
                _kcalAssunte.value = _kcalAssunte.value!! + (diario.
   chiloCalorieCena + diario.chiloCalorieColazione + diario.
   chiloCaloriePranzo + diario.chiloCalorieSpuntino)
                _grCarboidrati.value = _grCarboidrati.value!! + diario.
   carboidratiTot
                _grProteine.value = _grProteine.value!! + diario.proteineTot
                _grGrassi.value = _grGrassi.value!! + diario.grassiTot
                var acqua = 0
                for(i in 0..7) {

                    if (diario.acqua[i]) {
                        _litriBevuti.value = _litriBevuti.value!! + 0.25
                        acqua +=1
                    }
                    if (acqua == 8) _giorniAcqua.value = _giorniAcqua.value
   !! + 1
                }

            }
            _getStatistiche.value = true
        }
    }
```

Package: esercizio
All'interno di questo package troviamo tutte le classi usate per la gestione e salvataggio degli esercizi.

EsercizioAddPrefActivity: è la classe che visualizza i dettagli dell'esercizio selezionato nelle ricerche. Contiene i pulsanti per aggiungere al diario e alla lista dei preferiti.

EsercizioRDUActivity: è la classe che permette di modificare, eliminare e leggere gli esercizi già selezionati.

EsercizioViewModel: è la classe che collega le interfacce e il database. Recupera gli esercizi svolti, esercizi preferiti e quelli personalizzati dal database di ciascun utente.

MainActivity
E' l'activity che si apre all'avvio dell'applicazione e carica la schermata di InizioActivity.

Classi speciali

Package: retrofit
All'interno di questo package troviamo due interfacce e due classi per generare istanze retrofit. Le interfacce sono state utilizzate per dichiarare i metodi per eseguire le chiamate all'API. Ne troviamo due perché una è per l'API di Edamam (ApiInterface), che permette di fare le chiamate per ottenere i cibi, mentre la seconda interfaccia (ApiEserciziInterface) serve per fare le chiamate all'API Ninjas, quella responsabile degli esercizi.
Call è un metodo di Retrofit che consente di mandare una richiesta ad un webserver ed ottenere una risposta del tipo indicato tra parentesi angolari. La risposta dell'API sarà un Json che si va a mappare in classi Kotlin grazie all'utilizzo di Moshi. Le classi utilizzate per tale scopo sono raccolte nella cartella model/json_parsing.
Le classi Istance sono delle classi che hanno un attributo che implementa una delle due interfacce a seconda dell'API a cui ci si deve collegare. Ad esempio RetrofitEserciziInstance è un object che ha come attributi api_esercizi, che implementa ApiEserciziInterface, questo permette di accedere al metodo GetEsercizi, che è il metodo vero e proprio per eseguire il GET dei dati.

```
interface ApiInterface {
    //Call    un metodo di Retrofit che consente di mandare una richiesta ad
    un webserver ed ottenere una risposta.
    @GET("/api/food-database/v2/parser")
    fun getFoodFromNameOrUPC(@Query("app_id") idApp : String,
                             @Query("app_key") keyApp : String,
                             @Query("ingr") ingredient : String,
                             @Query("upc") upc : String): Call<Json_FoodList
    >

}
```

```
interface ApiEserciziInterface {


    @GET("/v1/caloriesburned")
    fun getEsercizi(@Header("X-Api-Key") key : String,
                    @Query("activity") activity : String): Call<List<
    Esercizio>>
}
```

Package: utils
Contiene la classe APICredentials la quale contiene gli URL utilizzati per le chiamate all'API e le API keys.

Parti di codice interessanti

```
/* Questa serie di funzioni sono quelle che permettono di gestire l'
   animazione dei bicchieri all'interno del diario
La prima funzione ovvero startAnimation    quella che fa partire l'
   animazione del riempimento del bicchiere
La seconda funzione    quella che controlla che controlla i bicchieri che
   devono essere riempiti al momento dell'apertura
della pagina del diario e fa partire l'animazione. Se i bicchieri riempiti
   sono ad esempio i alla fine setta l' i-esimo + 1
con l'immagine del bicchiere vuoto con il + dentro ad indicare che pu
   essere cliccato per aumentare la quantit  di acqua bevuta
La terza funzione invece ovvero onClickGlass    il 'cuore' del funzionamento
    dinamico dei bicchieri
essa associa ad ogni immagine del bicchiere un listener che fa la seguente
   operazione
- Se l'i-esimo bicchiere    pieno e viene cliccato allora esso dovr
   diventare vuoto con un + al suo interno
e tutti i bicchieri dopo di lui dovranno invece essere settatti con l'
   immagine del bicchiere vuoto
- Se invece l'i-esimo bicchiere    vuoto e viene cliccato allora il
   bicchiere dopo di lui dovr  essere settato con il simbolo +
mentre tutti quelli prima di lui dovranno fare lo start dell'animazione
   riempiendosi

La quarta funzione invece    quella che si occupa di andare ad aggiornare l'
   etichetta in cui andiamo
a visualizzare la quantit  totale di acqua bevuta e nel caso di
   raggiungimento di una quantit  pari
a due litri verr  visualizzato un messaggio di congratulazioni
*/


private fun startAnimation(glass : ImageView){
        glass.setBackgroundResource(R.drawable.filling_animation)
        val frameAnimation: AnimationDrawable = glass.background as
   AnimationDrawable
        frameAnimation.start()
    }

    fun checkFullGlasses(){
        for(i in 0..7) {
            if (model.diario.value!!.acqua[i]) {
                acqua[i] = model.diario.value!!.acqua[i]
                startAnimation(glasses[i])
                if(i < 7)
                    glasses[i+1].setBackgroundResource(R.drawable.
   empty_glass_plus)
            }
        }
        checkAcquaBevuta()
    }

    private fun onClickGlass(){
        for(i in 0..7){
            glasses[i].setOnClickListener {
                if(acqua[i]){
                    glasses[i].setBackgroundResource(R.drawable.
   empty_glass_plus)
                    for(x in i..7){
```

```
                        if(x<7)
                            glasses[x+1].setBackgroundResource(R.drawable.
    empty_glass)
                        acqua[x] = false
                    }
                }else {
                    for(x in 0..i){
                        startAnimation(glasses[x])
                        acqua[x] = true
                    }
                    if(i < 7){
                        glasses[i+1].setBackgroundResource(R.drawable.
    empty_glass_plus)
                    }
                }
                checkAcquaBevuta()
            }
        }
    }

    private fun checkAcquaBevuta(){
        var litri_acqua = 0.0
        for (bicchiere in acqua)
            if(bicchiere)
                litri_acqua += 0.25
        model.setAcqua(litri_acqua)
        /*Per fare in modo che il messaggio di congratulazioni venga
    mostrato una sola volta nel caso di raggiungimento
        dell'obiettivo controllo che il totale sia a 2 litri, se ho gia
    mostrato il messaggio nella stesso ciclo di vita del fragment
        e se il settimo bicchiere d'acqua era gia presente al momento del
    caricamento del diario
         */
        if(litri_acqua == 2.0){
            binding.imageViewGoldMedal.isVisible = true
            if(!flag_congratulazioni && !model.diario.value!!.acqua[7]) {
                flag_congratulazioni = true
                openCongratulazioni()
            }
        }else{
            binding.imageViewGoldMedal.isVisible = false
        }

    }
```

```
/* Questa classe    stata creata per poter andare a cambiare in maniera
    semplici i parametri di collegamento
all'API in modo da non dover andare a ricerca tra le varie classi i punti
    esatti in cui rimpiazzare questi dati */

class APICredentials {
    companion object {
        @JvmStatic
        val BASE_URL: String = "https://api.edamam.com"
        @JvmStatic
        val API_ID: String = "a4e62224"
        @JvmStatic
        val API_KEY: String = "2b0adbda47d1293d35d373d9f2ffcee1"
        @JvmStatic
        val BASE_URL_ESERCIZI: String = "https://api.api-ninjas.com"
```

```
        @JvmStatic
        val API_KEY_ESERCIZI: String = "WF3P9samgfmtcd0NlopT2w==
    OaTsIyDXQC9EvQKf"
     }
}
```

```
/*Classe che rappresenta l'utente a cui abbiamo fatto estendere l'
    interfaccia Parcelable in
maniera tale da poter passare un suo oggetto da un fragment ad un altro,
    questa soluzione ci ha
fatto molto comodo soprattutto in fase di registrazione dove ci saremmo
    trovati a dover passare un numero
sempre pi  elevato di parametri da un fragment all'altro, in quanto la
    registrazione effettiva dell'utente
viene fatta solamente al termine della navigazione quando tutti i dati dell'
    utente sono disponibili.
Con questa soluzione invece abbiamo passato sempre e solo un oggetto di tipo
     Utente che andavamo piano
piano a valorizzare. Ogni utente viene registrato con una dieta di default
    che    la cliamatica, ovvero
la dieta che ogni persona dovrebbe seguire per uno stile di vita sano.
    Abbiamo comunque lasciato la
possibilit  all'utente di modioficare la propria dieta nell'apposito
    fragment Diete */

@Parcelize
data class Utente(
        var nome: String,
        var cognome: String,
        var email: String,
        var LAF: Double,
        var agonistico: Boolean,
        var sesso: String,
        var data_nascita: String,
        var altezza: Int,
        var peso_attuale: Double,
        var sport: String?,
        var dieta: String

) : Parcelable {    constructor(): this("","","",0.0,false,"","",0,0.0,"","
    Climatica")
```

```
/* Questo pezzo di codice    quello che si occupa di gestire un bottone che
    fa cose diverse in base
in base a chi attiva l'apertura del Dialog, infatti l'utente pu  aprire il
    dialog che conteine
questo bottone in due modi, uno    quello di clicccare sul bottone '+' dei
    personalizzati, oppure cliccando su un
item della recycler view dei personalizzati.
Nel primo caso il bottone si dovr  occupare di aggiungere al DB dei
    personalizzati il
pasto dopo aver controllato che i valori immessi nei campi siano diversi da
    "" (stringa vuota) e 0.0.
Nel secondo caso invece il bottone dovr  gestire la modifica del pasto
    selezionato della
recycler view, al primo click su di esso dovr  rendere visibile la form di
    modifica mentre dal secondo click in
poi dovr  controllare che l'utente abbia effettivamente cambiato almeno un
    campo del prodotto
in tal caso andr  a modificare il pasto nel db */
```

```
11
12 var flag=false
13         btnAggiungiLista.setOnClickListener {
14             layout_info.visibility = View.VISIBLE
15             layout_quantita.visibility = View.GONE
16             var kcal_salva = kcal.text.toString()
17             var carbo_salva = carbo.text.toString()
18             var proteine_salva = proteine.text.toString()
19             var grassi_salva = grassi.text.toString()
20             var titolo = titolo.text.toString().trim()
21             if(kcal_salva != "" && carbo_salva != "" && proteine_salva != ""
     && grassi_salva != "" && titolo != "" &&
22                 kcal_salva.toDouble() != 0.0 && (carbo_salva.toDouble() !=
    0.0 || proteine_salva.toDouble() != 0.0 && grassi_salva.toDouble() !=
    0.0)) {
23                 if(bottone == "clickItem"){
24                     if (flag && valueAreChanged(position,kcal_salva,
    carbo_salva, proteine_salva,grassi_salva,titolo)){
25                         model.updatePersonalizzatoOnDB(
26                             id,requireArguments().getString("bottone")!!,
    titolo, kcal_salva.toDouble(),
27                             proteine_salva.toDouble(), carbo_salva.toDouble
    (), grassi_salva.toDouble(), requireContext())
28                         flag=false
29                         dialogLayout.visibility = View.GONE
30                     }else{
31                         Toast.makeText(requireContext(),"Cambia i valori
    prima di salvare",Toast.LENGTH_LONG).show()
32                         flag=true
33                     }
34                 }else {
35                     model.setPersonalizzatiOnDB(
36                         requireArguments().getString("bottone")!!, titolo,
    kcal_salva.toDouble(),
37                         proteine_salva.toDouble(), carbo_salva.toDouble(),
    grassi_salva.toDouble(), requireContext()
38                     )
39                     dialogLayout.visibility = View.GONE
40                 }
41                 model.getPersonalizzati(requireArguments().getString("
    bottone")!!)
42             }
43             else
44                 Toast.makeText(requireContext(),"Per favore completa tutti i
     campi o modifica i valori prima di salvare",Toast.LENGTH_LONG).show()
45         }
```

```
/* Questo pezzo di codice    quello che gestisce l'aggiunta del prodotto dai
    preferiti al
diario la soluzione interessante trovata    stata quella di usare un flag
   per gestire la
pressione dello stesso bottone che al primo click deve solamente eseguire la
visualizzazione del layout all'interno del quale l'utene pu  andare ad
   inserire la
quantit  mentre dal secondo click in poi deve andare a controllare che la
   quantit
sia diversa da 0 o da "" (stringa vuota) in tal caso pu  aggiungere il
   pasto al diario */

var flag = false
        dialog.btnAddDiario.setOnClickListener {
            dialog.layout_quantita.visibility = View.VISIBLE
            val quantita = dialog.editTextQuantita.text.toString().toDouble
   ()
            if(quantita != 0.0 && quantita.toString() != "") {
                model.setPastoOnDB(requireArguments().getString("bottone")
   !!,model.preferitiLiveData.value!![position].id,
                    model.preferitiLiveData.value!![position].image,model.
   preferitiLiveData.value!![position].nome,
                    model.preferitiLiveData.value!![position].calorie,model.
   preferitiLiveData.value!![position].proteine,
                    model.preferitiLiveData.value!![position].carboidrati,
   model.preferitiLiveData.value!![position].grassi,
                    quantita,requireContext())
                dialog.dismiss()
            }
            else {
                if (flag)
                    Toast.makeText(requireContext(),"Per favore inserisci
   una quantit  diversa da $quantita se desideri aggiungere il prodotto al
   Diario",Toast.LENGTH_LONG).show()
                flag = true
            }
        }
```

3.6 Testing

Abbiamo effettuato due tipologie di TEST: il primo su JUnit (Local Unit Test) ed il secondo con Espresso (Functional UI Test).

Local Unit Test

Per quanto riguarda il primo test, ci siamo soffermati sul calcolo del fabbisogno giornaliero di calorie, in quanto la nostra applicazione si incentra tutto su di esso.

Sono stati creati 6 diversi Test per far vedere come l'algoritmo restituisca il fabbisogno in maniera coerente e secondo i dati precisi inseriti dall'utente.

```
1  package com.example.fittracker
2
3  import com.example.fittracker.model.Utente
4  import org.junit.Assert
5  import org.junit.Before
6  import org.junit.Test
7
8  class FabbisognoTest {
9
10     @Before
11     fun setUp(){
12
13     }
14
15     // Test per vedere se il metodo calculateFabbisogno calcola
       correttamente le calorie giornaliere
16
17     @Test
18     fun check_CorrectFabbisogno(){
19         val fabbisogno = Utente()
20         val utente = Utente("Alessandro","Rongoni","alerong@gmail.com"
       ,1.725,true,"Uomo","2000-05-18",183,76.0,"Calcio","Climatica")
21         Assert.assertEquals(3230, fabbisogno.calculateFabbisogno(utente.
       data_nascita,utente.sesso,utente.peso_attuale,utente.altezza,utente.LAF))
22     }
23
24     @Test
25     fun check_UncorrectFabbisogno1(){
26         val fabbisogno = Utente()
27         val utente = Utente("Alessandro","Rongoni","alerong@gmail.com"
       ,1.725,true,"Uomo","2000-05-18",183,76.0,"Calcio","Climatica")
28         Assert.assertNotEquals(3229, fabbisogno.calculateFabbisogno(utente.
       data_nascita,utente.sesso,utente.peso_attuale,utente.altezza,utente.LAF))
29     }
30
31     @Test
32     fun check_UncorrectFabbisogno2(){
33         val fabbisogno = Utente()
34         val utente = Utente("Alessandro","Rongoni","alerong@gmail.com"
       ,1.725,true,"Uomo","2000-05-18",183,76.0,"Calcio","Climatica")
35         Assert.assertNotEquals(3231, fabbisogno.calculateFabbisogno(utente.
       data_nascita,utente.sesso,utente.peso_attuale,utente.altezza,utente.LAF))
36     }
37
38
39
```

```
40
41
42
43
44
45      @Test
46      fun check_LAFSedentarioFabbisogno(){
47          val fabbisogno = Utente()
48          val utente = Utente("Alessandro","Rongoni","alerong@gmail.com",1.2,
    true,"Uomo","2000-05-18",183,76.0,"Calcio","Climatica")
49          Assert.assertEquals(2247, fabbisogno.calculateFabbisogno(utente.
    data_nascita,utente.sesso,utente.peso_attuale,utente.altezza,utente.LAF))
50      }
51
52      @Test
53      fun check_DonnaFabbisogno(){
54          val fabbisogno = Utente()
55          val utente = Utente("Martina","Rossi","martina.rossiOF.com",1.55,
    false,"Donna","2002-06-23",167,55.0,"","Climatica")
56          Assert.assertEquals(1239, fabbisogno.calculateFabbisogno(utente.
    data_nascita,utente.sesso,utente.peso_attuale,utente.altezza,utente.LAF))
57      }
58
59      @Test
60      fun check_DonnaCambioEtaFabbisogno(){
61          val fabbisogno = Utente()
62          val utente = Utente("Martina","Rossi","martina.rossiOF.com",1.55,
    false,"Donna","1999-06-23",167,55.0,"","Climatica")
63          Assert.assertNotEquals(1239, fabbisogno.calculateFabbisogno(utente.
    data_nascita,utente.sesso,utente.peso_attuale,utente.altezza,utente.LAF))
64      }
65
66
67  }
```

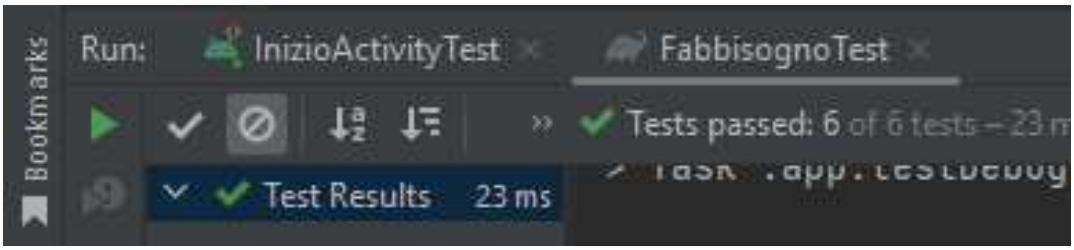


Figura 11: Risultato del test JUnit.

Svolgere questo test ci è stato molto utile, in quanto ci siamo accorti che il calcolo del fabbisogno non avveniva in modo corretto. Il motivo per cui questo succedeva era perché una condizione non era mai verificata, e ciò comportava uno squilibrio nelle calorie finali.

Functional UI Test

Il secondo ed ultimo test va a verificare il corretto funzionamento dell'UI, simulando delle azioni che l'utente può svolgere.

L'Activity che abbiamo deciso di testare è la InizioActivity, in quanto tutto parte da essa. Abbiamo simulato la parte di registrazione e di autenticazione dell'utente.

```
package com.example.fittracker.home

import android.widget.DatePicker
import androidx.lifecycle.Lifecycle
import androidx.test.core.app.ActivityScenario
import androidx.test.core.app.launchActivity
import androidx.test.espresso.Espresso
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions
import androidx.test.espresso.action.ViewActions.click
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.contrib.PickerActions
import androidx.test.espresso.matcher.ViewMatchers.*
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.filters.LargeTest
import com.example.fittracker.R
import com.example.fittracker.autenticazione.InizioActivity
import org.hamcrest.Matchers
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith


@LargeTest
@RunWith(AndroidJUnit4::class)
internal class InizioActivityTest{
    private lateinit var scenario: ActivityScenario<InizioActivity>

    @Before
    fun setUp(){
        scenario = launchActivity()
        scenario.moveToState(Lifecycle.State.RESUMED)
    }


    /*============================================================
                        ELIMINARE I DATI E SVUOTARE LA CACHE PRIMA DI EFFETTURARE IL TEST
        Il cellulare mantiene i dati d'accesso e il LoginTest fallirebbe in quanto non trova le view di input
        per email e password
        ============================================================*/
    @Test
    fun registerTest(){

        // TEST DI REGISTRAZIONE in questo caso l'utente    gi   registrato e non permette la registrazione
        onView(withId(R.id.btInizia)).perform(click())
        onView(withId(R.id.rB_sedentario)).perform(click())
        onView(withId(R.id.sB_agonistico)).perform(click())
        onView(withId(R.id.imageView16)).check(matches(isDisplayed()))
        onView(withId(R.id.bt_AvantiObb)).perform(click())
        onView(withId(R.id.rB_uomo)).perform(click())
        onView(withId(R.id.bt_AvantiSesso)).perform(click())
        Espresso.pressBack()
        onView(withId(R.id.rB_donna)).perform(click())
        onView(withId(R.id.bt_AvantiSesso)).perform(click())
        onView(withText("Dati Personali")).check(matches(isDisplayed()))
        onView(withId(R.id.tv_dataNascita)).perform(click())
```

```
        val year = 2000
        val month = 11
        val day = 15
        onView(withClassName(Matchers.equalTo(DatePicker::class.java.name)))
    .perform(PickerActions.setDate(year,month,day))
        onView(withId(android.R.id.button1)).perform(click())
        onView(withId(R.id.tE_name)).perform(ViewActions.typeText("Giovanna"
    ))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.tE_surname)).perform(ViewActions.typeText("Rossi"
    ))
        onView(withId(R.id.tE_name)).check(matches(withText("Giovanna")))
        onView(withId(R.id.tE_surname)).check(matches(withText("Rossi")))
        onView(withId(R.id.tv_dataNascita)).check(matches(withText("
    15-11-2000")))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.bt_AvantiDati)).perform(click())
        onView(withId(R.id.eT_altezza)).perform(ViewActions.typeText("30"))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.bt_AvantiAltezza)).perform(click())
        onView(withId(R.id.eT_altezza)).check(matches(withText("")))
        onView(withId(R.id.eT_altezza)).perform(ViewActions.typeText("165"))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.bt_AvantiAltezza)).perform(click())
        onView(withId(R.id.eT_PesoAttuale)).perform(ViewActions.typeText("29
    "))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.bt_AvantiPesoAttuale)).perform(click())
        onView(withId(R.id.eT_PesoAttuale)).check(matches(withText("")))
        onView(withId(R.id.eT_PesoAttuale)).perform(ViewActions.typeText("56
    "))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.bt_AvantiPesoAttuale)).perform(click())
        onView(withId(R.id.cB_calcio)).perform(click())
        onView(withId(R.id.bt_AvantiPesoObb)).perform(click())
        onView(withId(R.id.btnRegister)).check(matches(withText("Registrati"
    )))
        onView(withId(R.id.InputEmail)).perform(ViewActions.typeText("
    test@espresso.it"))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.InputPassword)).perform(ViewActions.typeText("
    123456"))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.InputCorrectPassword)).perform(ViewActions.
    typeText("123456"))
        Espresso.closeSoftKeyboard()
        onView(withId(R.id.btnRegister)).perform(click())



     }
```

```
106
107     @Test
108     fun LoginTest(){
109         /*=====================================================
110         SUCCESSO nel caso in cui non si ha una cache da ripulire e nessun
    dato di accesso memorizzato
111         ============================================================*/
112
113         onView(withId(R.id.btAccesso)).check(matches(withText("ACCEDI")))
114         onView(withId(R.id.btAccesso)).perform(click())
115         onView(withId(R.id.InputEmailLogin)).perform(ViewActions.typeText("
    test@espresso.it"))
116         Espresso.closeSoftKeyboard()
117         onView(withId(R.id.InputPasswordLogin)).perform(ViewActions.typeText
    ("123456"))
118         Espresso.closeSoftKeyboard()
119         onView(withId(R.id.btnLogin)).perform(click())
120     }
121
122
123 }
```

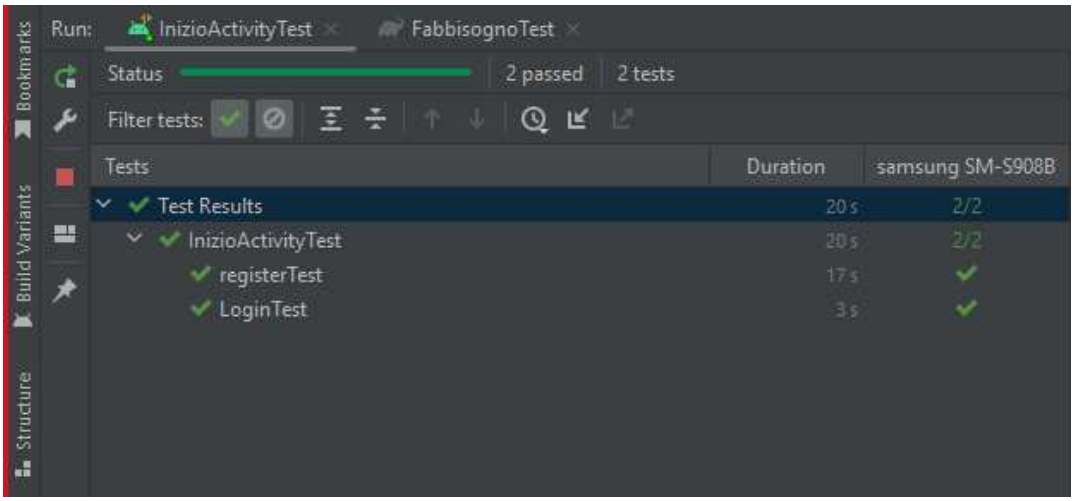


Figura 12: Risultato del test Espresso.

In questi due test di verificano i vari spostamenti tra le facciate della registrazione e dell'accesso, si verificano che i valori sbagliati inseriti vengano resettati al click sul pulsante, si verifica che gli input messi rispecchiano quello che la vista mostra (ad esempio il DataPicker, nome e cognome) e infine si verifica che tornando indietro i valori vengano resettati.

3.7 Conclusioni, Known Bugs e Future Features

Un bug che abbiamo riscontrato riguarda l'aggiornamento delle progressBar dei carboidrati, proteine e dei grassi alla modifica della dieta. Infatti, i valori massimi vengono aggiornati correttamente in base alla tipologia di dieta, ma la visualizzazione della lunghezza della progressBar rimane invariata. Cliccando su un qualsiasi pulsante di pasto/esercizio, la lunghezza delle barre viene aggiornata.
Implementazioni future potrebbero riguardare nuove statistiche da controllare e funzionalità relative al fitness.

4 Parte II: Implementazione di funzioni in Android (Kotlin)

Questa è la seconda parte della relazione, dedicata all'implementazione di modifiche interessanti riguardanti l'applicazione. Tali funzioni aggiunte sono state fatte da Alessandro Rongoni e sono oggetto di tirocinio e tesi di quest'ultimo.

4.1 Assistente vocale

Come prima funzionalità ho voluto aggiungere un assistente vocale all'applicazione.
Per fare ciò ho valutato diverse librerie/frameworks di assistenti vocali, per poi scegliere "Alan Studio" [7]. Ciò che mi ha convinto ad utilizzare tale piattaforma è stata la sua completa documentazione, la grande community connessa e la flessibilità e robustezza.

Come funziona Alan?

Alan è una piattaforma AI vocale avanzata che consente di aggiungere un'interfaccia vocale alla nostra app senza sovraccarico.

Alan è una piattaforma AI conversazionale[1] end-to-end per creare assistenti vocali e chatbot in-app robusti e affidabili. Non è necessario creare modelli di lingua parlata, addestrare il software per il riconoscimento vocale, distribuire e ospitare componenti vocali: il backend Alan AI svolge la maggior parte del lavoro. Con Alan, un'esperienza vocale per l'app può essere creata e sviluppata da un singolo sviluppatore, piuttosto che da un team di esperti di Machine Learning e Dev Ops.

Alan ci consente di andare oltre le capacità delle interfacce touch e di digitazione ed abilitare la voce a qualsiasi flusso di lavoro o funzione complessa nella nostra app. Gli script vocali sono scritti in **JavaScript**, il che li rende altamente personalizzabili e flessibili.

Le interfacce vocali create con Alan vengono create una volta e distribuite ovunque, quindi non sarà necessario ricostruirle per piattaforme specifiche. Alan offre SDK leggeri da integrare con:

- **Framework Web:** React, Angular, Vue, Ember, JavaScript, Electron;
- **iOS:** Swift e Objective-C;
- **Android:** Kotlin e Java;
- **Framework multipiattaforma:** Flutter, Ionic, React Native, Apache Cordova;

Se dovessi apportare modifiche all'interfaccia vocale, posso eseguire il push di tali modifiche senza dover rilasciare una nuova versione dell'app. Grazie all'ambiente serverless di Alan, qualsiasi modifica all'interfaccia vocale viene resa immediatamente disponibile agli utenti.

[1] In informatica, modalità di funzionamento di un computer, consistente in una serie di domande e di risposte fra un utente e il computer stesso.

Implementazione

Come prima cosa ho creato una nuova activity all'interno dell'applicazione dove accogliere il codice per la gestione dell'assistente vocale ed aggiunto la sua dipendenza all'interno del file "build.gradle".

```
1  package com.example.fittracker.funzioni
2
3  import androidx.appcompat.app.AppCompatActivity
4  import android.os.Bundle
5  import android.util.Log
6  import com.alan.alansdk.AlanCallback
7  import com.alan.alansdk.AlanConfig
8  import com.alan.alansdk.button.AlanButton
9  import com.alan.alansdk.events.EventCommand
10 import com.example.fittracker.R
11 import org.json.JSONException
12 //Activity per la gestione dell'assistente vocale
13 class AssistenteActivity : AppCompatActivity() {
14
15     private var alanButton: AlanButton? = null
16
17
18     override fun onCreate(savedInstanceState: Bundle?) {
19         super.onCreate(savedInstanceState)
20 //setting dell'interfaccia dell'assistente vocale
21         setContentView(R.layout.activity_assistente)
22         /// Defining the project key
23         val config = AlanConfig.builder().setProjectId("455
    d65a3089f659eca63f5cfd13cfc502e956eca572e1d8b807a3e2338fdd0dc/stage").
    build()
24         alanButton = findViewById(R.id.alan_button)
25         alanButton?.initWithConfig(config)
26
27         val alanCallback: AlanCallback = object : AlanCallback() {
28             /// Handling commands from Alan Studio
29             override fun onCommand(eventCommand: EventCommand) {
30                 try {
31                     val command = eventCommand.data
32                     val commandName = command.getJSONObject("data").
    getString("command")
33                     Log.d("AlanButton", "onCommand: commandName:
    $commandName")
34                 } catch (e: JSONException) {
35                     e.message?.let { Log.e("AlanButton", it) }
36                 }
37             }
38         };
39
40         /// Registering callbacks
41         alanButton?.registerCallback(alanCallback);
42     }
43 }
```

Successivamente ho creato un nuovo progetto all'interno di Alan Studio, il quale mi ha permesso di collegare il mio codice tramite la SDK Key che troviamo all'interno della variabile config a riga 23. Infine mi è bastato creare un file "voice script" personalizzato per far capire all'assistente vocale le domande e come rispondere correttamente ad esse.
All'interno della console in Alan Studio è possibile modificare diverse impostazioni, come ad

esempio il bottone per avviare Alan, la voce e la lingua dell'assistente (purtroppo a pagamento) ed altre funzioni utili allo sviluppatore per migliorare la performance del voice assistant, come l'attivazione dell'assistente tramite una parola chiave e il suo avvio in background.

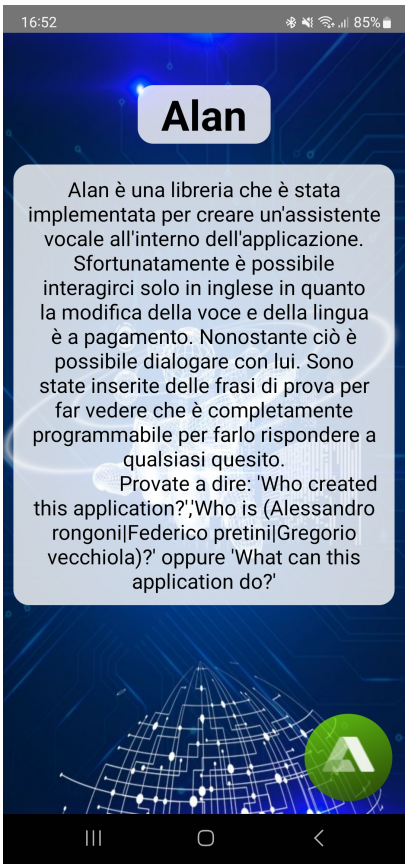


Figura 13: Interfaccia dell'activity per la gestione e l'interazione dell'assistente vocale

Gli scripts in cui vengono scritte le domande e le risposte che l'assistente dovrà elaborare sono in Javascript. Alan Studio mette a disposizione degli scripts di esempio con delle domande e risposte giusto per parlare ed altre un po' più elaborate come il Calculators e News, che permettono di svolgere azioni più complesse rispetto ad una semplice conversazione.

Conclusioni e modifiche future

L'implementazione di un'assistente vocale può aiutare l'utente nell'ambito del fitness, ad esempio per tenere il tempo di un esercizio oppure per il recupero, automatizzando ciò che l'atleta dovrebbe fare manualmente tramite l'orologio o un timer/cronometro. Un miglioramento futuro potrebbe sicuramente riguardare nuovi comandi vocali ed il cambiamento della lingua, rendendo più facile l'interazione tra uomo e assistente.

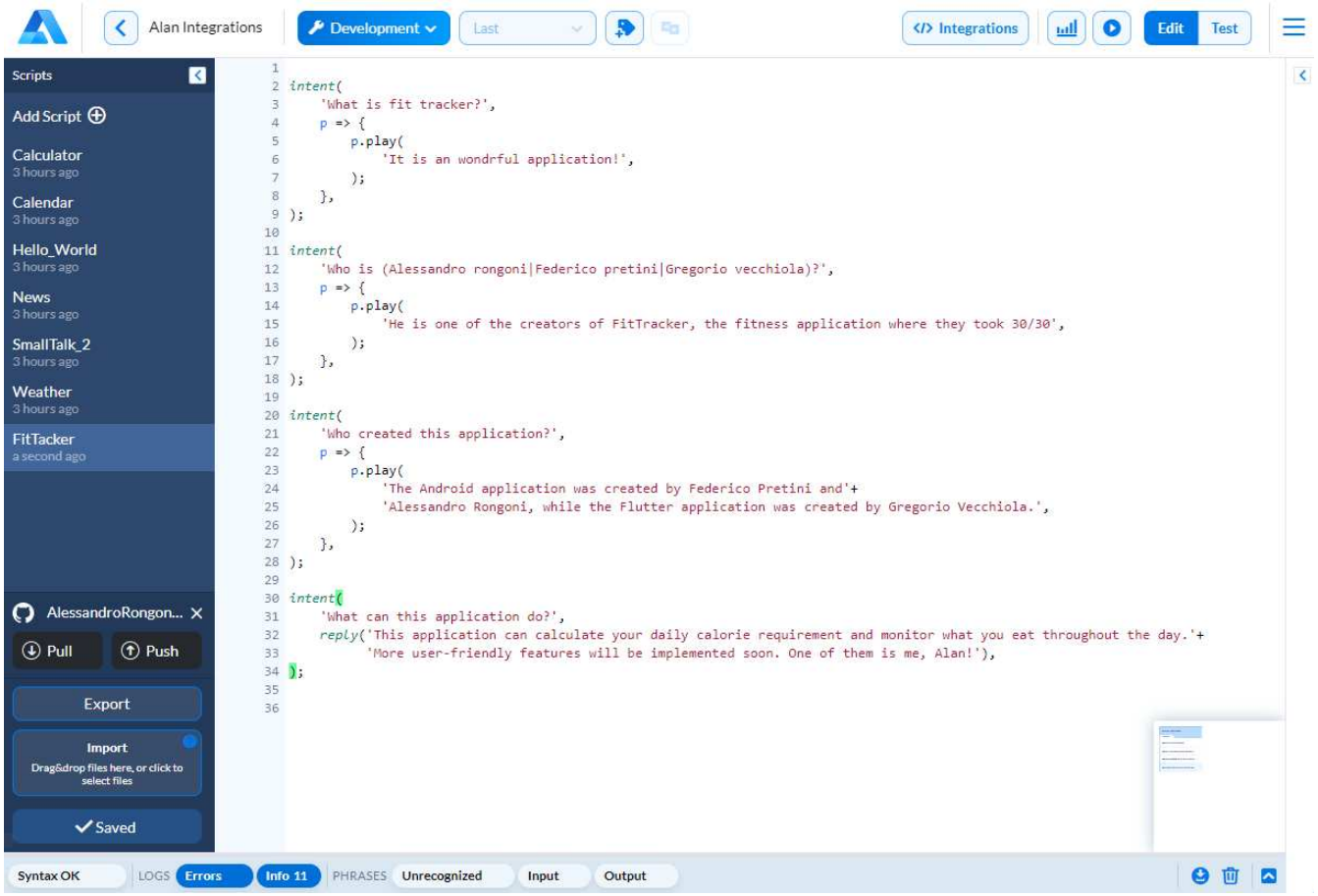


Figura 14: Console di Alan Studio per la scrittura degli scripts.

4.2 Pose Detector (ML Kit)

ML Kit [9] offre agli sviluppatori di dispositivi mobili le competenze di Google relative al machine learning in un pacchetto potente e facile da utilizzare. Rende le app per iOS e Android più coinvolgenti, personalizzate e utili con soluzioni ottimizzate per l'esecuzione sui dispositivi. Combinano i migliori modelli di machine learning con pipeline di elaborazione avanzate e li offrono tramite **API** facili da usare per Vision e Natural Language per abilitare potenti casi d'uso nelle app, nel nostro caso utilizzeremo l'API per il **rilevamento della posa**.

L'API ML Kit Pose Detection [10] è una soluzione leggera e versatile che consente agli sviluppatori di app di rilevare la posa del corpo di un soggetto in tempo reale da un video continuo o da un'immagine statica. Una posa descrive la posizione del corpo in un momento con un insieme di punti di riferimento scheletrici. I punti di riferimento corrispondono a diverse parti del corpo, come spalle e fianchi. Le posizioni relative dei punti di riferimento possono essere utilizzate per distinguere una posa da un'altra.

Questa API è disponibile in versione beta, il che significa che potrebbe essere modificata in modi incompatibili con le versioni precedenti e non è soggetta a SLA (accordo sul livello del servizio) o norme sul ritiro.

ML Kit Pose Detection [10] produce una corrispondenza scheletrica a tutto il corpo di 33 punti che include punti di riferimento del viso (orecchie, occhi, bocca e naso) e punta su mani e piedi. La figura 16 mostra i punti di riferimento che guardano l'utente attraverso la videocamera, pertanto si tratta di un'immagine speculare. Il lato destro dell'utente viene visualizzato a sinistra dell'immagine:

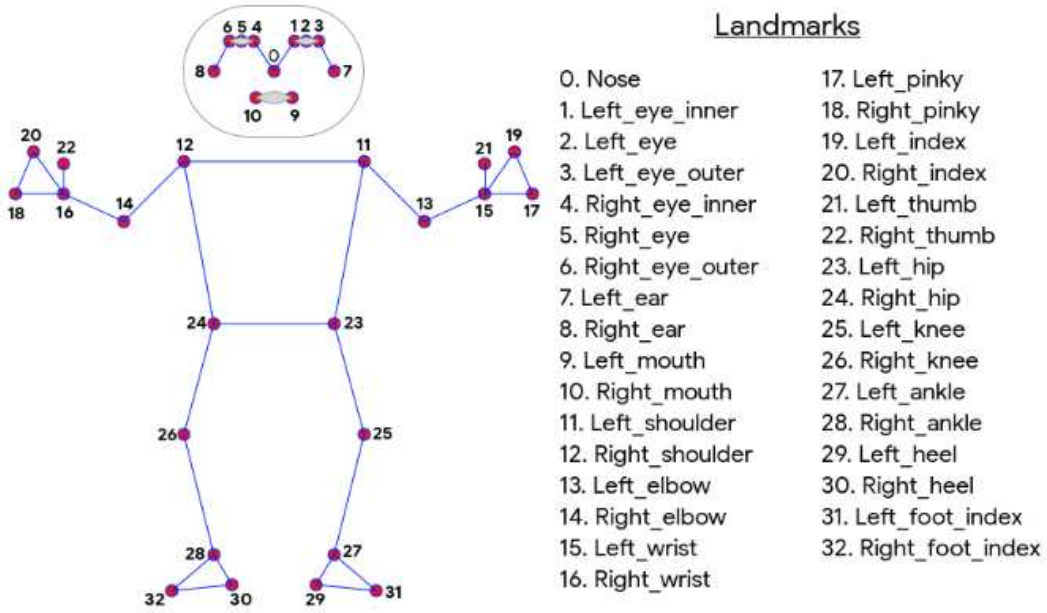


Figura 15: Punti di riferimento.

ML Kit Pose Detection [10] non richiede attrezzature specializzate o competenze nel machine learning per ottenere ottimi risultati. Con questa tecnologia gli sviluppatori possono creare esperienze uniche per i propri utenti con solo poche righe di codice.

Per rilevare una posa, deve essere presente il volto dell'utente. Il rilevamento della posizione funziona meglio quando l'intero corpo del soggetto è visibile nell'inquadratura, ma rileva anche

la posa parziale del corpo. In tal caso, vengono assegnate coordinate al di fuori dell'immagine ai punti di riferimento non riconosciuti.

Funzionalità chiave

I principali motivi per cui ho deciso di provare ed implementare ML Kit Pose Detection [10] sono i seguenti:

- **Assistenza multipiattaforma** Usufruisci della stessa esperienza sia su Android che su iOS.
- **Monitoraggio completo del corpo** Il modello restituisce 33 punti di riferimento scheletrici chiave, tra cui le posizioni di mani e piedi.
- **Punteggio InFrameLikelihood** per ogni punto di riferimento, una misura che indica la probabilità che il punto di riferimento si trovi all'interno del frame dell'immagine. Il punteggio ha un intervallo compreso tra 0,0 e 1,0, dove 1,0 indica un'elevata affidabilità.
- **Due SDK ottimizzati:** l'SDK di base viene eseguito in tempo reale su telefoni moderni come Pixel 4 e iPhone X. Restituisce i risultati a una velocità di circa 30 e circa 45 f/s. Tuttavia, la precisione delle coordinate di riferimento può variare. L'SDK preciso restituisce risultati a una frequenza fotogrammi più lenta, ma produce valori di coordinate più precisi.
- **Coordinata Z per l'analisi della profondità** Questo valore può aiutare a determinare se parti del corpo degli utenti si trovano davanti o dietro i fianchi. Nella sezione *Coordinata Z* di seguito è possibile entrare più nel dettaglio per avere ulteriori informazioni.

L'API Pose Detection è simile all'**API Facial Recognition** in quanto restituisce un insieme di punti di riferimento e la loro posizione. Tuttavia, mentre il rilevamento dei volti cerca di riconoscere anche caratteristiche come la bocca sorridente o gli occhi aperti, la funzionalità di posa non associa alcun significato ai punti di riferimento in una posa o alla posa stessa. Inoltre è possibile creare i tuoi algoritmi per interpretare una posa personalizzata.

La funzionalità di rilevamento della posizione può rilevare una sola persona in un'immagine. Se nell'immagine sono presenti due persone, il modello assegnerà punti di riferimento alla persona rilevata con la massima affidabilità.

Coordinata Z

La coordinata Z è un valore sperimentale calcolato per ogni punto di riferimento. Viene misurato in "pixel immagine" come le coordinate X e Y, ma non è un valore 3D reale. L'asse Z è perpendicolare alla videocamera e passa tra i fianchi di un soggetto. L'origine dell'asse Z è indicativamente del punto centrale tra i fianchi (sinistra/destra e fronte/retro rispetto alla fotocamera). I valori Z negativi si riferiscono alla fotocamera; i valori positivi non lo sono. La coordinata Z non ha un limite superiore o inferiore.

Risultati di esempio

La tabella seguente mostra le coordinate e l'InFrameLikelihood per alcuni punti di riferimento nella posa a destra. Le coordinate Z della mano sinistra dell'utente sono negative, poiché si trovano di fronte al centro del soggetto e rivolte verso la fotocamera.

Punto di riferimento	Tipo	Posizione	Probabilità di integrazione nei frame
11	LEFT_SHOULDER	(734.9671, 550.7924, -118.11934)	0,9999038
12	DESTRA_SHOULDER	(391.27032, 583.2485, -321.15836)	0,9999894
13	TITOLO_SINISTRA	(903.83704, 754.676, -219.67009)	0,9836427
14	TITOLO_DESTRO	(322.18152, 842.5973, -179.28519)	0,99970156
15	PRIMO_SCRITTO	(1073.8956, 654.9725, -820.93463)	0,9737737
16	PRIMO_SCRITTO	(218.27956, 1015.70435, -683.6567)	0,995568
17	LEFT_PINKY	(1146.1635, 609.6432, -956.9976)	0,95273364
18	DESTRA_PINKY	(176.17755, 1065.838, -776.5006)	0,9785348

Figura 16: Esempio di rilevamento del corpo (Esempio tratto dal sito di ML Kit).

Implementazione su Android

Come prima cosa mi sono assicurato che nel file **build.gradle** a livello di progetto fosse incluso il repository Maven di Google nelle sezioni *buildscript* e *allprojects*. Successivamente ho aggiunto le dipendenze per le librerie Android di ML Kit al file gradle a livello di app del modulo, ovvero **app/build.gradle**:

```
dependencies {
  // If you want to use the base sdk
  implementation 'com.google.mlkit:pose-detection:18.0.0-beta3'
  // If you want to use the accurate sdk
  implementation 'com.google.mlkit:pose-detection-accurate:18.0.0-beta3'
}
```

1. Creazione di un'istanza di PoseDetector

Per rilevare una posa in un'immagine, bisogna prima creare un'istanza di **PoseDetector** e, facoltativamente, specificare le impostazioni del rilevatore.

Modalità di rilevamento

PoseDetector opera in due modalità di rilevamento. Ovvero:

- **STREAM_MODE (valore predefinito)**
  Il rilevatore di posa rileva innanzitutto la persona più in evidenza nell'immagine, quindi esegue il rilevamento. Nei frame successivi, il passaggio di rilevamento di una persona non verrà eseguito a meno che la persona non venga oscurata o non venga più rilevata con un'elevata affidabilità. Il rilevatore di posizioni proverà a tenere traccia della persona più in evidenza e a restituire la sua posa a ogni inferenza. Questo riduce la latenza e agevola il rilevamento. Questa modalità è perfetta qualora si volesse rilevare la posa in un video stream (noi useremo questa modalità).
- **STREAM_IMAGE_MODE**
  Il rilevatore di posa rileva la persona e poi esegue il rilevamento della posa. Il passaggio di rilevamento persone viene eseguito per ogni immagine, quindi la latenza è superiore e non è previsto il monitoraggio delle persone. Si usa questa modalità quando utilizziamo il rilevamento della posa su immagini statiche o dove non vogliamo il monitoraggio.

Configurazione hardware

PoseDetector supporta più configurazioni hardware per l'ottimizzazione delle prestazioni:

- **CPU:** eseguo il rilevatore utilizzando solo la CPU.
- **CPU_GPU:** eseguo il rilevatore utilizzando CPU e GPU.

Quando si crea le opzioni del rilevatore, possiamo utilizzare l'API **setPreferredHardware-Configs** per controllare la selezione dell'hardware. Per impostazione predefinita, tutte le configurazioni hardware sono impostate come preferite.

ML Kit prenderà in considerazione disponibilità, stabilità, correttezza e latenza di ogni configurazione e sceglierà la migliore dalle configurazioni preferite. Se nessuna delle configurazioni preferite è applicabile, la configurazione **CPU** verrà utilizzata automaticamente come riserva. ML Kit eseguirà questi controlli e la relativa preparazione in modo non bloccante prima di abilitare qualsiasi accelerazione, quindi è molto probabile che la prima volta che eseguiremo il rilevatore, si utilizzerà CPU. Al termine della preparazione, la configurazione migliore verrà utilizzata nelle esecuzioni successive.

Esempi di utilizzo di **setPreferredHardwareConfigs**:

- Per consentire al ML Kit di scegliere la configurazione migliore, non bisogna chiamare questa API.
- Se non vogliamo abilitare alcuna accelerazione, passiamo solo CPU.
- Se vogliamo utilizzare la GPU per alleggerire la CPU anche se potrebbe essere più lenta, passiamo solo in **CPU_GPU**.

Specifichiamo le opzioni del rilevatore di posizioni:

```
// Base pose detector with streaming frames, when depending on the pose-
    detection sdk
val options = PoseDetectorOptions.Builder()
    .setDetectorMode(PoseDetectorOptions.STREAM_MODE)
    .build()

// Accurate pose detector on static images, when depending on the pose-
    detection-accurate sdk
val options = AccuratePoseDetectorOptions.Builder()
    .setDetectorMode(AccuratePoseDetectorOptions.SINGLE_IMAGE_MODE)
    .build()
```

Infine, creiamo un'istanza di **PoseDetector**. Passiamo le opzioni specificate:

```
val poseDetector = PoseDetection.getClient(options)
```

2. Prepariamo l'immagine di input

Per rilevare le posizioni in un'immagine, creiamo un oggetto **InputImage** da un elemento Bitmap, media.Image, ByteBuffer, da un array di byte o da un file sul dispositivo.
Passiamo quindi l'oggetto **InputImage** a **PoseDetector**.

Per il rilevamento della posa, bisogna utilizzare un'immagine con dimensioni di almeno **480 x 360** pixel. Se, invece, rileviamo le posizioni in tempo reale, l'acquisizione di frame con questa risoluzione minima può aiutarci a ridurre la latenza.

Possiamo creare un oggetto **InputImage** da origini diverse, ognuno dei quali è illustrato nelle sezioni successive.

Con media.Image

Per creare un oggetto InputImage da un oggetto **media.Image**, ad esempio quando acquisiamo un'immagine dalla fotocamera di un dispositivo, passiamo l'oggetto media.Image e la rotazione dell'immagine a InputImage.fromMediaImage().

Se utilizziamo la libreria **CameraX**, le classi OnImageCapturedListener e ImageAnalysis.Analyzer calcolano il valore di rotazione automaticamente.

```
private class YourImageAnalyzer : ImageAnalysis.Analyzer {

    override fun analyze(imageProxy: ImageProxy) {
        val mediaImage = imageProxy.image
        if (mediaImage != null) {
            val image = InputImage.fromMediaImage(mediaImage, imageProxy.
    imageInfo.rotationDegrees)
            // Pass image to an ML Kit Vision API
            // ...
        }
    }
}
```

Se non si utilizza una raccolta di fotocamere che ci dà il grado di rotazione dell'immagine, possiamo calcolarla in base al grado di rotazione e all'orientamento del sensore della fotocamera del dispositivo:

```
private val ORIENTATIONS = SparseIntArray()

```

```
init {
    ORIENTATIONS.append(Surface.ROTATION_0, 0)
    ORIENTATIONS.append(Surface.ROTATION_90, 90)
    ORIENTATIONS.append(Surface.ROTATION_180, 180)
    ORIENTATIONS.append(Surface.ROTATION_270, 270)
}

/**
 * Get the angle by which an image must be rotated given the device's current
 * orientation.
 */
@RequiresApi(api = Build.VERSION_CODES.LOLLIPOP)
@Throws(CameraAccessException::class)
private fun getRotationCompensation(cameraId: String, activity: Activity, isFrontFacing: Boolean): Int {
    // Get the device's current rotation relative to its "native" orientation.
    // Then, from the ORIENTATIONS table, look up the angle the image must be
    // rotated to compensate for the device's rotation.
    val deviceRotation = activity.windowManager.defaultDisplay.rotation
    var rotationCompensation = ORIENTATIONS.get(deviceRotation)

    // Get the device's sensor orientation.
    val cameraManager = activity.getSystemService(CAMERA_SERVICE) as CameraManager
    val sensorOrientation = cameraManager
            .getCameraCharacteristics(cameraId)
            .get(CameraCharacteristics.SENSOR_ORIENTATION)!!

    if (isFrontFacing) {
        rotationCompensation = (sensorOrientation + rotationCompensation) % 360
    } else { // back-facing
        rotationCompensation = (sensorOrientation - rotationCompensation + 360) % 360
    }
    return rotationCompensation
}
```

Quindi, trasmettiamo l'oggetto **media.Image** e il valore del grado di rotazione a **InputImage.fromMediaImage()**:

```
val image = InputImage.fromMediaImage(mediaImage, rotation)
```

Usando un URI di file

Per creare un oggetto InputImage da un URI del file, passiamo il contesto dell'app e l'URI del file a **InputImage.fromFilePath()**. Ciò è utile quando utilizziamo un intent **ACTION_GET_CONTENT** per richiedere all'utente di selezionare un'immagine dalla sua app Galleria.

```
val image: InputImage
try {
    image = InputImage.fromFilePath(context, uri)
} catch (e: IOException) {
    e.printStackTrace()
}
```

Con ByteBuffer o ByteArray

Per creare un oggetto InputImage da un elemento **ByteBuffer** o **ByteArray**, innanzitutto calcoliamo il grado di rotazione delle immagini come descritto in precedenza per l'input media.Image. Quindi, creiamo l'oggetto InputImage con il buffer o l'array, insieme all'altezza, alla larghezza, al formato di codifica dei colori e del grado di rotazione dell'immagine:

```
val image = InputImage.fromByteBuffer(
        byteBuffer,
        /* image width */ 480,
        /* image height */ 360,
        rotationDegrees,
        InputImage.IMAGE_FORMAT_NV21 // or IMAGE_FORMAT_YV12
)
// Or:
val image = InputImage.fromByteArray(
        byteArray,
        /* image width */ 480,
        /* image height */ 360,
        rotationDegrees,
        InputImage.IMAGE_FORMAT_NV21 // or IMAGE_FORMAT_YV12
)
```

Con bitmap

Per creare un oggetto InputImage da un oggetto Bitmap, effettuiamo la seguente dichiarazione:

```
val image = InputImage.fromBitmap(bitmap, 0)
```

L'immagine è rappresentata da un oggetto **Bitmap** insieme a gradi di rotazione.

3. Elaborazione dell'immagine

Passiamo l'oggetto InputImage, preparato nella sezione precedente, al metodo **process** di **PoseDetector**.

```
Task<Pose> result = poseDetector.process(image)
        .addOnSuccessListener { results ->
            // Task completed successfully
            // ...
        }
        .addOnFailureListener { e ->
            // Task failed with an exception
            // ...
        }
```

Se si utilizza l'API CameraX, bisogna assicurarsi di chiudere ImageProxy quando la usiamo, ad esempio aggiungendo un elemento OnCompleteListener a Task restituito dal metodo process.

4. Ricevere informazioni sulla posa rilevata

Se viene rilevata una persona nell'immagine, l'API di rilevamento della posa restituisce un oggetto **Pose** con 33 **PoseLandmark**.

Se la persona non si trovava completamente all'interno dell'immagine, il modello assegna le coordinate mancanti dei punti di riferimento al di fuori del frame e assegna valori bassi di InFrameConfidence.

Se non è stata rilevata alcuna persona nel frame, l'oggetto **Pose** non contiene **PoseLandmark**.

```
// Get all PoseLandmarks. If no person was detected, the list will be empty
val allPoseLandmarks = pose.getAllPoseLandmarks()

// Or get specific PoseLandmarks individually. These will all be null if no
   person
// was detected
val leftShoulder = pose.getPoseLandmark(PoseLandmark.LEFT_SHOULDER)
val rightShoulder = pose.getPoseLandmark(PoseLandmark.RIGHT_SHOULDER)
val leftElbow = pose.getPoseLandmark(PoseLandmark.LEFT_ELBOW)
val rightElbow = pose.getPoseLandmark(PoseLandmark.RIGHT_ELBOW)
val leftWrist = pose.getPoseLandmark(PoseLandmark.LEFT_WRIST)
val rightWrist = pose.getPoseLandmark(PoseLandmark.RIGHT_WRIST)
val leftHip = pose.getPoseLandmark(PoseLandmark.LEFT_HIP)
val rightHip = pose.getPoseLandmark(PoseLandmark.RIGHT_HIP)
val leftKnee = pose.getPoseLandmark(PoseLandmark.LEFT_KNEE)
val rightKnee = pose.getPoseLandmark(PoseLandmark.RIGHT_KNEE)
val leftAnkle = pose.getPoseLandmark(PoseLandmark.LEFT_ANKLE)
val rightAnkle = pose.getPoseLandmark(PoseLandmark.RIGHT_ANKLE)
val leftPinky = pose.getPoseLandmark(PoseLandmark.LEFT_PINKY)
val rightPinky = pose.getPoseLandmark(PoseLandmark.RIGHT_PINKY)
val leftIndex = pose.getPoseLandmark(PoseLandmark.LEFT_INDEX)
val rightIndex = pose.getPoseLandmark(PoseLandmark.RIGHT_INDEX)
val leftThumb = pose.getPoseLandmark(PoseLandmark.LEFT_THUMB)
val rightThumb = pose.getPoseLandmark(PoseLandmark.RIGHT_THUMB)
val leftHeel = pose.getPoseLandmark(PoseLandmark.LEFT_HEEL)
val rightHeel = pose.getPoseLandmark(PoseLandmark.RIGHT_HEEL)
val leftFootIndex = pose.getPoseLandmark(PoseLandmark.LEFT_FOOT_INDEX)
val rightFootIndex = pose.getPoseLandmark(PoseLandmark.RIGHT_FOOT_INDEX)
val nose = pose.getPoseLandmark(PoseLandmark.NOSE)
val leftEyeInner = pose.getPoseLandmark(PoseLandmark.LEFT_EYE_INNER)
val leftEye = pose.getPoseLandmark(PoseLandmark.LEFT_EYE)
val leftEyeOuter = pose.getPoseLandmark(PoseLandmark.LEFT_EYE_OUTER)
val rightEyeInner = pose.getPoseLandmark(PoseLandmark.RIGHT_EYE_INNER)
val rightEye = pose.getPoseLandmark(PoseLandmark.RIGHT_EYE)
val rightEyeOuter = pose.getPoseLandmark(PoseLandmark.RIGHT_EYE_OUTER)
val leftEar = pose.getPoseLandmark(PoseLandmark.LEFT_EAR)
val rightEar = pose.getPoseLandmark(PoseLandmark.RIGHT_EAR)
val leftMouth = pose.getPoseLandmark(PoseLandmark.LEFT_MOUTH)
val rightMouth = pose.getPoseLandmark(PoseLandmark.RIGHT_MOUTH)
```

Suggerimenti per migliorare il rendimento

La qualità dei risultati dipende dalla qualità dell'immagine di input:

- Affinché ML Kit possa rilevare con precisione la posizione, la persona nell'immagine deve essere rappresentata da dati pixel sufficienti; per prestazioni ottimali, il soggetto deve essere di almeno 256 x 256 pixel.
- Se rileviamo la posa in un'applicazione in tempo reale, si può anche valutare le dimensioni complessive delle immagini di input. Le immagini più piccole possono essere elaborate più velocemente, quindi per ridurre la latenza, acquisiamo immagini a risoluzioni più basse, ma bisogna tenere presenti i requisiti di risoluzione precedenti e assicurarsi che il soggetto occupi il maggior numero possibile di immagini.
- Una scarsa messa a fuoco dell'immagine può anche influire sulla precisione. Se non si ottiene risultati accettabili, chiediamo all'utente di riottenere l'immagine.

Se vogliamo utilizzare il rilevamento della posa in un'applicazione in tempo reale (il nostro caso), seguiamo queste linee guida per ottenere migliori frequenze fotogrammi:

- Usiamo l'SDK per il rilevamento della posizione di base e **STREAM_MODE**.
- Potremmo acquisire immagini a una risoluzione inferiore. Tuttavia, teniamo presente anche i requisiti relativi alle dimensioni delle immagini di questa API.
- Se utilizziamo l'API **Camera** o **camera2**, limitiamo le chiamate al rilevatore. Se è disponibile un nuovo frame video mentre il rilevatore è in esecuzione, trasciniamo il frame.
- Se utilizziamo l'API **CameraX**, bisogna assicurarsi che la strategia di contro-pressione sia impostata sul suo valore predefinito **ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST**. Garantisce che venga pubblicata una sola immagine alla volta. Se vengono generate più immagini quando lo strumento di analisi è occupato, le immagini vengono eliminate automaticamente e non vengono messe in coda per la pubblicazione. Una volta che l'immagine da analizzare viene chiusa chiamando ImageProxy.close(), verrà pubblicata l'ultima immagine successiva.
- Se utilizziamo l'output del rilevatore per sovrapporre la grafica sull'immagine di input, otteniamo prima il risultato dal ML Kit, quindi visualizziamo l'immagine e l'overlay in un solo passaggio. Viene visualizzato sulla superficie di visualizzazione una sola volta per ogni frame di input.
- Se utilizziamo l'API Camera2, acquisiamo le immagini nel formato **ImageFormat.YUV_420_888**. Se utilizziamo la precedente API Camera, acquisiamo le immagini nel formato **ImageFormat.NV21**.

Pose Detector all'interno del progetto FitTracker

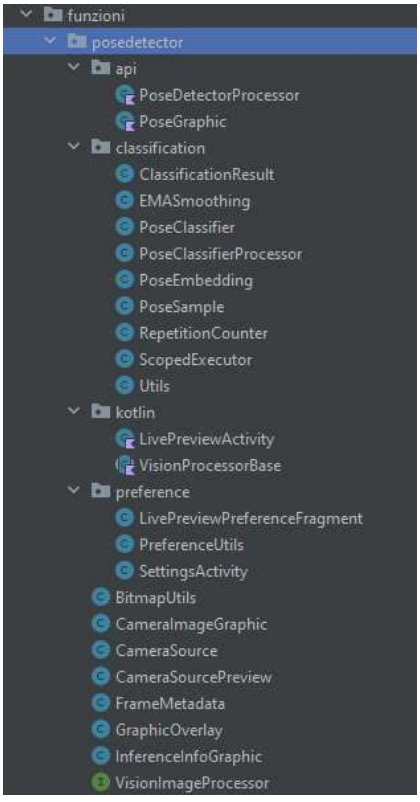


Figura 17: Il package contenente tutte le classi usate per l'implementazione del Pose Detector.

All'interno di "funzioni", ho suddiviso tutte le classi che mi servivano in altri package, ovvero "api" (la quale contiene l'istanza del PoseDetector vera e propria), "classification" (contenente le classi utili per il riconoscimento della posa, la loro classificazione ed il conteggio), "kotlin" (le classi kotlin, contenente l'activity che ospita il codice per la creazione della preview della camera e la gestione dei permessi), "preference" (si occupa della gestione dei settings, delle preferenze dell'utente e preferenze varie), ed infine tutte le classi[2] contenenti i metodi per far funzionare la camera e gestire l'overlay dei PoseLandmark.

PoseDetectorProcessor Class

```
1  package com.example.fittracker.funzioni.posedetector.api
2
3  import android.content.Context
4  import android.util.Log
5  import com.example.fittracker.funzioni.posedetector.GraphicOverlay
6  import com.example.fittracker.funzioni.posedetector.classification.
       PoseClassifierProcessor
7  import com.example.fittracker.funzioni.posedetector.kotlin.
       VisionProcessorBase
8  import com.google.android.gms.tasks.Task
9  import com.google.android.odml.image.MlImage
10 import com.google.mlkit.vision.common.InputImage
11 import com.google.mlkit.vision.pose.Pose
12 import com.google.mlkit.vision.pose.PoseDetection
13 import com.google.mlkit.vision.pose.PoseDetector
14 import com.google.mlkit.vision.pose.PoseDetectorOptionsBase
15 import java.util.ArrayList
16 import java.util.concurrent.Executor
17 import java.util.concurrent.Executors
18
19 /** A processor to run pose detector. */
20 class PoseDetectorProcessor(
21   private val context: Context,
22   options: PoseDetectorOptionsBase,
23   private val showInFrameLikelihood: Boolean,
24   private val visualizeZ: Boolean,
25   private val rescaleZForVisualization: Boolean,
26   private val runClassification: Boolean,
27   private val isStreamMode: Boolean
28 ) : VisionProcessorBase<PoseDetectorProcessor.PoseWithClassification>(
       context) {
29
30   private val detector: PoseDetector
31   private val classificationExecutor: Executor
32
33   private var poseClassifierProcessor: PoseClassifierProcessor? = null
34
35   /** Internal class to hold Pose and classification results. */
36   class PoseWithClassification(val pose: Pose, val classificationResult:
       List<String>)
37
38   init {
39     detector = PoseDetection.getClient(options)
40     classificationExecutor = Executors.newSingleThreadExecutor()
41   }
```

[2]Queste classi verrano poi riutilizzate anche per il corretto funzionamento dell'API ImageLabeling implementata in seguito.

```
42
43  override fun stop() {
44    super.stop()
45    detector.close()
46  }
47
48  override fun detectInImage(image: InputImage): Task<PoseWithClassification
     > {
49    return detector
50      .process(image)
51      .continueWith(
52        classificationExecutor
53      ) { task ->
54        val pose = task.result
55        var classificationResult: List<String> = ArrayList()
56        if (runClassification) {
57          if (poseClassifierProcessor == null) {
58            poseClassifierProcessor =
59                PoseClassifierProcessor(
60                    context,
61                    isStreamMode
62                )
63          }
64          classificationResult = poseClassifierProcessor!!.getPoseResult(
     pose)
65        }
66        PoseWithClassification(pose, classificationResult)
67      }
68  }
69
70  override fun detectInImage(image: MlImage): Task<PoseWithClassification> {
71    return detector
72      .process(image)
73      .continueWith(
74        classificationExecutor
75      ) { task ->
76        val pose = task.result
77        var classificationResult: List<String> = ArrayList()
78        if (runClassification) {
79          if (poseClassifierProcessor == null) {
80            poseClassifierProcessor =
81                PoseClassifierProcessor(
82                    context,
83                    isStreamMode
84                )
85          }
86          classificationResult = poseClassifierProcessor!!.getPoseResult(
     pose)
87        }
88        PoseWithClassification(pose, classificationResult)
89      }
90  }
91
92  override fun onSuccess(
93    results: PoseWithClassification,
94    graphicOverlay: GraphicOverlay
95  ) {
96    graphicOverlay.add(
97      PoseGraphic(
```

```
        graphicOverlay,
        results.pose,
        showInFrameLikelihood,
        visualizeZ,
        rescaleZForVisualization,
        results.classificationResult
      )
    )
  }

  override fun onFailure(e: Exception) {
    Log.e(TAG, "Pose detection failed!", e)
  }

  override fun isMlImageEnabled(context: Context?): Boolean {
    // Use MlImage in Pose Detection by default, change it to OFF to switch to InputImage.
    return true
  }

  companion object {
    private const val TAG = "PoseDetectorProcessor"
  }
}
```

PoseGrapich Class

```
package com.example.fittracker.funzioni.posedetector.api

import android.graphics.Canvas
import android.graphics.Color
import android.graphics.Paint
import com.example.fittracker.funzioni.posedetector.GraphicOverlay
import com.google.mlkit.vision.pose.Pose
import com.google.mlkit.vision.pose.PoseLandmark
import java.util.Locale

/** Draw the detected pose in preview. */
class PoseGraphic
internal constructor(
  overlay: GraphicOverlay,
  private val pose: Pose,
  private val showInFrameLikelihood: Boolean,
  private val visualizeZ: Boolean,
  private val rescaleZForVisualization: Boolean,
  private val poseClassification: List<String>
) : GraphicOverlay.Graphic(overlay) {
  private var zMin = java.lang.Float.MAX_VALUE
  private var zMax = java.lang.Float.MIN_VALUE
  private val classificationTextPaint: Paint = Paint()
  private val leftPaint: Paint
  private val rightPaint: Paint
  private val whitePaint: Paint

  init {
    classificationTextPaint.color = Color.WHITE
    classificationTextPaint.textSize = POSE_CLASSIFICATION_TEXT_SIZE
    classificationTextPaint.setShadowLayer(5.0f, 0f, 0f, Color.BLACK)

    whitePaint = Paint()
```

```
    whitePaint.strokeWidth = STROKE_WIDTH
    whitePaint.color = Color.WHITE
    whitePaint.textSize = IN_FRAME_LIKELIHOOD_TEXT_SIZE
    leftPaint = Paint()
    leftPaint.strokeWidth = STROKE_WIDTH
    leftPaint.color = Color.GREEN
    rightPaint = Paint()
    rightPaint.strokeWidth = STROKE_WIDTH
    rightPaint.color = Color.YELLOW
  }

  override fun draw(canvas: Canvas) {
    val landmarks = pose.allPoseLandmarks
    if (landmarks.isEmpty()) {
      return
    }

    // Draw pose classification text.
    val classificationX = POSE_CLASSIFICATION_TEXT_SIZE * 0.5f
    for (i in poseClassification.indices) {
      val classificationY =
        canvas.height -
          (POSE_CLASSIFICATION_TEXT_SIZE * 1.5f * (poseClassification.size -
    i).toFloat())
      canvas.drawText(
        poseClassification[i],
        classificationX,
        classificationY,
        classificationTextPaint
      )
    }

    // Draw all the points
    for (landmark in landmarks) {
      drawPoint(canvas, landmark, whitePaint)
      if (visualizeZ && rescaleZForVisualization) {
        zMin = zMin.coerceAtMost(landmark.position3D.z)
        zMax = zMax.coerceAtLeast(landmark.position3D.z)
      }
    }

    val nose = pose.getPoseLandmark(PoseLandmark.NOSE)
    val lefyEyeInner = pose.getPoseLandmark(PoseLandmark.LEFT_EYE_INNER)
    val lefyEye = pose.getPoseLandmark(PoseLandmark.LEFT_EYE)
    val leftEyeOuter = pose.getPoseLandmark(PoseLandmark.LEFT_EYE_OUTER)
    val rightEyeInner = pose.getPoseLandmark(PoseLandmark.RIGHT_EYE_INNER)
    val rightEye = pose.getPoseLandmark(PoseLandmark.RIGHT_EYE)
    val rightEyeOuter = pose.getPoseLandmark(PoseLandmark.RIGHT_EYE_OUTER)
    val leftEar = pose.getPoseLandmark(PoseLandmark.LEFT_EAR)
    val rightEar = pose.getPoseLandmark(PoseLandmark.RIGHT_EAR)
    val leftMouth = pose.getPoseLandmark(PoseLandmark.LEFT_MOUTH)
    val rightMouth = pose.getPoseLandmark(PoseLandmark.RIGHT_MOUTH)

    val leftShoulder = pose.getPoseLandmark(PoseLandmark.LEFT_SHOULDER)
    val rightShoulder = pose.getPoseLandmark(PoseLandmark.RIGHT_SHOULDER)
    val leftElbow = pose.getPoseLandmark(PoseLandmark.LEFT_ELBOW)
    val rightElbow = pose.getPoseLandmark(PoseLandmark.RIGHT_ELBOW)
    val leftWrist = pose.getPoseLandmark(PoseLandmark.LEFT_WRIST)
    val rightWrist = pose.getPoseLandmark(PoseLandmark.RIGHT_WRIST)
```

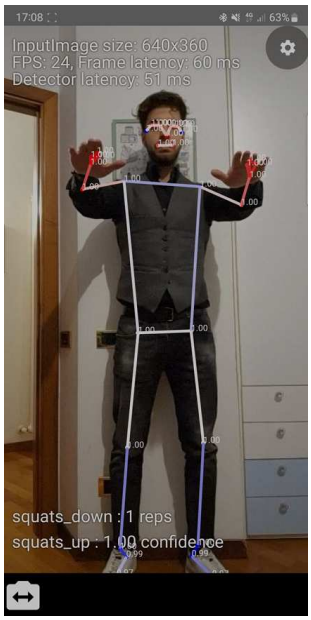
```
    val leftHip = pose.getPoseLandmark(PoseLandmark.LEFT_HIP)
    val rightHip = pose.getPoseLandmark(PoseLandmark.RIGHT_HIP)
    val leftKnee = pose.getPoseLandmark(PoseLandmark.LEFT_KNEE)
    val rightKnee = pose.getPoseLandmark(PoseLandmark.RIGHT_KNEE)
    val leftAnkle = pose.getPoseLandmark(PoseLandmark.LEFT_ANKLE)
    val rightAnkle = pose.getPoseLandmark(PoseLandmark.RIGHT_ANKLE)

    val leftPinky = pose.getPoseLandmark(PoseLandmark.LEFT_PINKY)
    val rightPinky = pose.getPoseLandmark(PoseLandmark.RIGHT_PINKY)
    val leftIndex = pose.getPoseLandmark(PoseLandmark.LEFT_INDEX)
    val rightIndex = pose.getPoseLandmark(PoseLandmark.RIGHT_INDEX)
    val leftThumb = pose.getPoseLandmark(PoseLandmark.LEFT_THUMB)
    val rightThumb = pose.getPoseLandmark(PoseLandmark.RIGHT_THUMB)
    val leftHeel = pose.getPoseLandmark(PoseLandmark.LEFT_HEEL)
    val rightHeel = pose.getPoseLandmark(PoseLandmark.RIGHT_HEEL)
    val leftFootIndex = pose.getPoseLandmark(PoseLandmark.LEFT_FOOT_INDEX)
    val rightFootIndex = pose.getPoseLandmark(PoseLandmark.RIGHT_FOOT_INDEX)

    // Face
    drawLine(canvas, nose, lefyEyeInner, whitePaint)
    drawLine(canvas, lefyEyeInner, lefyEye, whitePaint)
    drawLine(canvas, lefyEye, leftEyeOuter, whitePaint)
    drawLine(canvas, leftEyeOuter, leftEar, whitePaint)
    drawLine(canvas, nose, rightEyeInner, whitePaint)
    drawLine(canvas, rightEyeInner, rightEye, whitePaint)
    drawLine(canvas, rightEye, rightEyeOuter, whitePaint)
    drawLine(canvas, rightEyeOuter, rightEar, whitePaint)
    drawLine(canvas, leftMouth, rightMouth, whitePaint)

    drawLine(canvas, leftShoulder, rightShoulder, whitePaint)
    drawLine(canvas, leftHip, rightHip, whitePaint)

    // Left body
    drawLine(canvas, leftShoulder, leftElbow, leftPaint)
    drawLine(canvas, leftElbow, leftWrist, leftPaint)
    drawLine(canvas, leftShoulder, leftHip, leftPaint)
    drawLine(canvas, leftHip, leftKnee, leftPaint)
    drawLine(canvas, leftKnee, leftAnkle, leftPaint)
    drawLine(canvas, leftWrist, leftThumb, leftPaint)
    drawLine(canvas, leftWrist, leftPinky, leftPaint)
    drawLine(canvas, leftWrist, leftIndex, leftPaint)
    drawLine(canvas, leftIndex, leftPinky, leftPaint)
    drawLine(canvas, leftAnkle, leftHeel, leftPaint)
    drawLine(canvas, leftHeel, leftFootIndex, leftPaint)

    // Right body
    drawLine(canvas, rightShoulder, rightElbow, rightPaint)
    drawLine(canvas, rightElbow, rightWrist, rightPaint)
    drawLine(canvas, rightShoulder, rightHip, rightPaint)
    drawLine(canvas, rightHip, rightKnee, rightPaint)
    drawLine(canvas, rightKnee, rightAnkle, rightPaint)
    drawLine(canvas, rightWrist, rightThumb, rightPaint)
    drawLine(canvas, rightWrist, rightPinky, rightPaint)
    drawLine(canvas, rightWrist, rightIndex, rightPaint)
    drawLine(canvas, rightIndex, rightPinky, rightPaint)
    drawLine(canvas, rightAnkle, rightHeel, rightPaint)
    drawLine(canvas, rightHeel, rightFootIndex, rightPaint)

    // Draw inFrameLikelihood for all points
```

```
    if (showInFrameLikelihood) {
      for (landmark in landmarks) {
        canvas.drawText(
          String.format(Locale.US, "%.2f", landmark.inFrameLikelihood),
          translateX(landmark.position.x),
          translateY(landmark.position.y),
          whitePaint
        )
      }
    }
  }

  private fun drawPoint(canvas: Canvas, landmark: PoseLandmark, paint: Paint
  ) {
    val point = landmark.position3D
    updatePaintColorByZValue(
      paint,
      canvas,
      visualizeZ,
      rescaleZForVisualization,
      point.z,
      zMin,
      zMax
    )
    canvas.drawCircle(translateX(point.x), translateY(point.y), DOT_RADIUS,
   paint)
  }

  private fun drawLine(
    canvas: Canvas,
    startLandmark: PoseLandmark?,
    endLandmark: PoseLandmark?,
    paint: Paint
  ) {
    val start = startLandmark!!.position3D
    val end = endLandmark!!.position3D

    // Gets average z for the current body line
    val avgZInImagePixel = (start.z + end.z) / 2
    updatePaintColorByZValue(
      paint,
      canvas,
      visualizeZ,
      rescaleZForVisualization,
      avgZInImagePixel,
      zMin,
      zMax
    )

    canvas.drawLine(
      translateX(start.x),
      translateY(start.y),
      translateX(end.x),
      translateY(end.y),
      paint
    )
  }

  companion object {
```

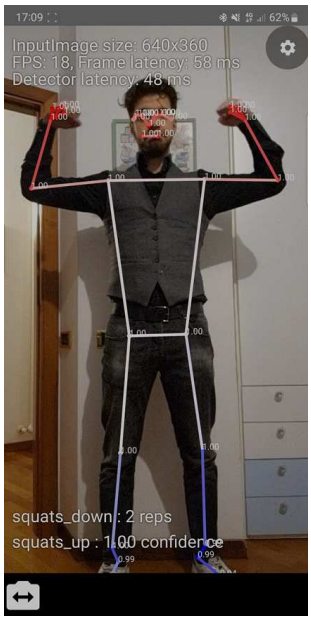
```
208
209     private const val DOT_RADIUS = 8.0f
210     private const val IN_FRAME_LIKELIHOOD_TEXT_SIZE = 30.0f
211     private const val STROKE_WIDTH = 10.0f
212     private const val POSE_CLASSIFICATION_TEXT_SIZE = 60.0f
213   }
214 }
```

Screenshot all'interno dell'app

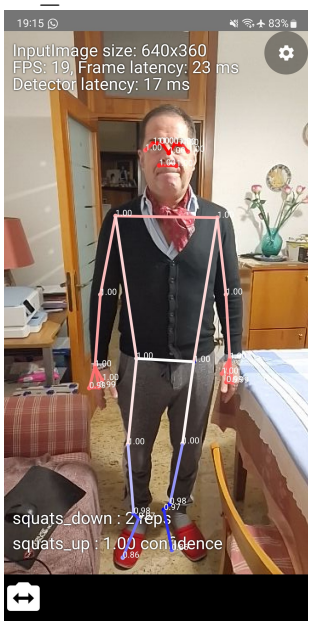
Riconoscimento dell'esercizio SQUAT_DOWN con valore di confidenza 0.57



Dopo aver superato un certo valore di confidenza, viene aumentato il contatore e riconosciuto l'esercizio SQUAT_UP con confidenza 1.00



Prove di pose detection



Cambio del soggetto di riferimento

4.3 Opzioni di classificazione delle posizioni

Con l'API ML Pose Detection [10], si possono ricavare interpretazioni significative di una posa controllando le posizioni relative delle varie parti del corpo. Io ho utilizzato la classificazione delle posizioni e conteggio delle ripetizioni con l'algoritmo k-NN, ma si può utilizzare anche per riconoscere una posizione di yoga oppure riconoscere dei gesti calcolando la distanza di un punto di riferimento.

Classificazione delle posizioni e conteggio delle ripetizioni con l'algoritmo k-NN

Una delle applicazioni più comuni per il rilevamento della posizione è il monitoraggio dell'attività fisica. Creare un classificatore di posa che riconosce specifiche posizioni di allenamento e conteggiare le ripetizioni può essere un'impresa impegnativa per gli sviluppatori.

In questa sezione viene descritto il modo in cui abbiamo creato un classificatore di pose personalizzato utilizzando **MediaPipe Colab** [11] e dimostri una categoria di lavoro funzionante nella nostra app FitTracker.

Per riconoscere le posizioni, utilizziamo l'algoritmo k-nearest neighbour (k-NN), perché è semplice e facile da usare. L'algoritmo determina la classe dell'oggetto in base ai campioni più vicini del set di addestramento.

Per creare e addestrare il riconoscimento:

1. Raccogliere esempi di immagini

Ho utilizzato dei campioni di immagini degli allenamenti target usate da ML Kit. Per ogni esercizio sono state scelte alcune centinaia di immagini[3], ad esempio "up" e "down", posizioni per flessioni. È importante raccogliere campioni che comprendano diverse angolazioni della videocamera, condizioni ambientali, forme del corpo e variazioni dell'allenamento.

Figura 18: Posizione delle flessioni in alto e in basso. (Immagine tratta dal sito di ML Kit)

[3]Per questo motivo aggiungere nuove pose può essere dispendioso, in quanto per far funzionare bene il riconoscimento della posa ci vuole un **Learning Set** di almeno 100 immagini per esercizio.

2. Eseguire il rilevamento della posa sulle immagini di esempio

Viene generato un set di punti di riferimento per la posa da utilizzare per l'addestramento. Non siamo interessati al rilevamento della posizione, dato che addestreremo il nostro modello nel passaggio successivo.

L'**algoritmo k-NN** scelto per la classificazione della posa personalizzata richiede una rappresentazione di funzionalità per ogni campione e una metrica per calcolare la distanza tra due vettori per trovare il target più vicino al campione di posa. Ciò significa che dobbiamo convertire i punti di riferimento della posa che abbiamo appena ottenuto.

Per convertire i punti di riferimento di una posa in un vettore caratteristica, utilizziamo le distanze a coppie tra elenchi predefiniti di articolazioni della posa, come la distanza tra il polso e la spalla, la caviglia e l'anca e i polsi sinistro e destro. Poiché la scala delle immagini può variare, bisogna normalizzare le posizioni per avere le stesse dimensioni e lo stesso orientamento del busto prima di convertire i punti di riferimento.

3. Addestrare il modello e conteggiare le ripetizioni

Viene utilizzato **MediaPipe Colab** [11] per accedere al codice per la categoria di classificazione e addestrare il modello.

Per conteggiare le ripetizioni, è stato utilizzato un altro algoritmo Colab per monitorare la soglia di probabilità di una posizione di posa target. Ad esempio:

- Quando la probabilità della classe di posa "down" supera una determinata soglia per la prima volta, l'algoritmo contrassegna la classe di posa "down".
- Quando la probabilità scende al di sotto della soglia, l'algoritmo stabilisce che la classe di posa "giù" è stata chiusa e aumenta il contatore.

Figura 19: Esempio di conteggio delle ripetizioni. (Immagine tratta dal sito di ML Kit)

4. Integrazione con l'app FitTracker

Colab genera un file CSV che possiamo completare con tutti gli esempi di posa. In questa sezione vedremo come integrare il file CSV con l'app creare FitTracker per Android per visualizzare la classificazione delle posizioni personalizzate in tempo reale.

Aggiungere il file CSV personalizzato

- Aggiungo il file CSV alla cartella asset dell'app.
- in **PoseClassifierProcessor**, bisogna aggiornare le variabili *POSE_SAMPLES_FILE* e *POSE_CLASSES* in modo che corrispondano al file CSV e agli esempi di posa.
- Creare ed eseguire l'app.

Bisogna tenere presente che la classificazione potrebbe non funzionare correttamente se non sono disponibili campioni in numero sufficiente. In genere, sono necessari circa 100 campioni per classe di posa [11].

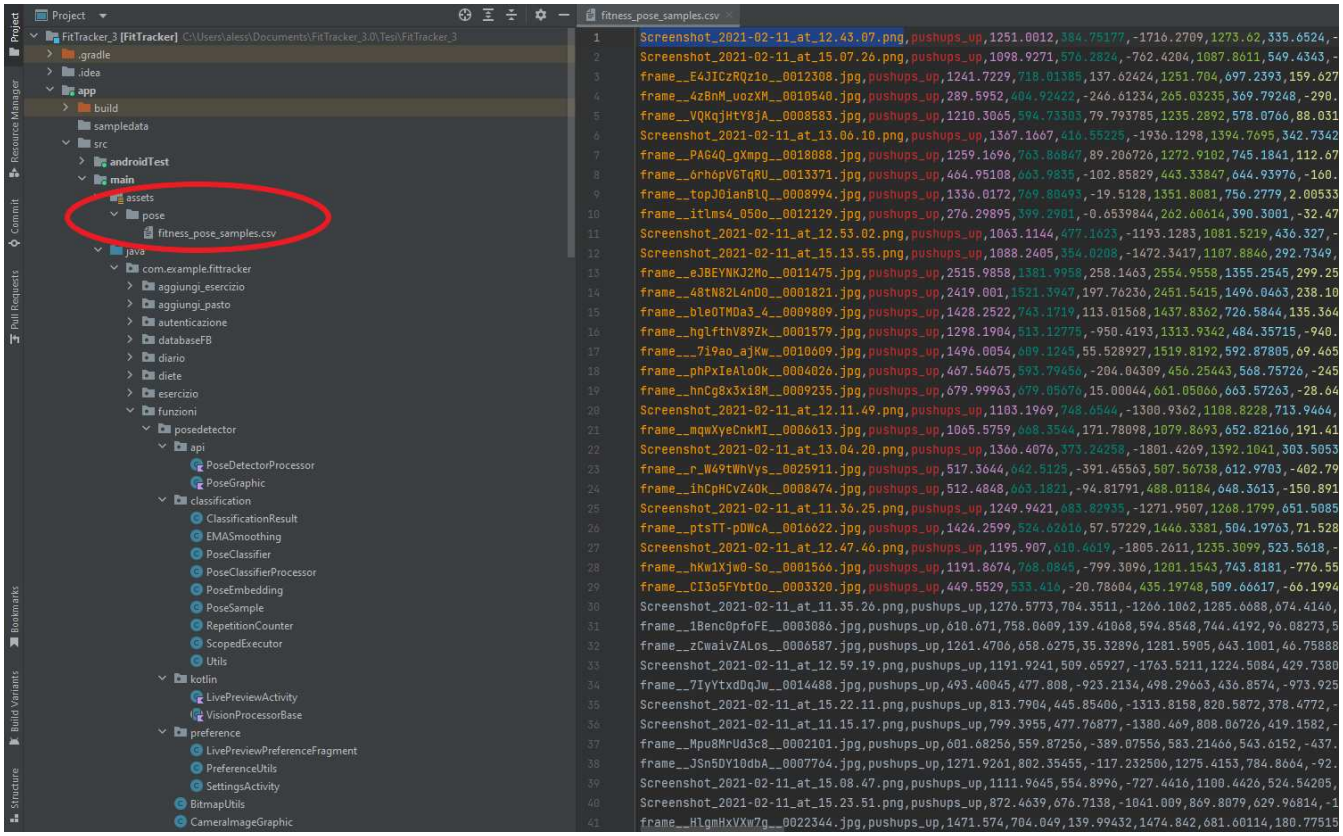


Figura 20: File CSV all'interno della cartella "assets" del nostro progetto.

Riconoscere semplici gesti calcolando la distanza di un punto di riferimento

Quando due o più punti di riferimento sono vicini tra loro, possono essere utilizzati per riconoscere i gesti. Ad esempio, quando il punto di riferimento per una o più dita su una mano è vicino al punto di riferimento per il naso, è probabile che l'utente stia toccando il suo volto.

Figura 21: Interpretare una posa. (Esempio tratto dal sito di ML Kit)

Riconoscere una posizione yoga con l'euristica angolare

Per individuare la posizione di uno yoga, bisogna calcolare gli angoli di varie articolazioni. Ad esempio, la Figura 22 di seguito mostra la posizione di yoga del Guerriero II. Gli angoli approssimativi che identificano questa posa sono indicati nella figura.

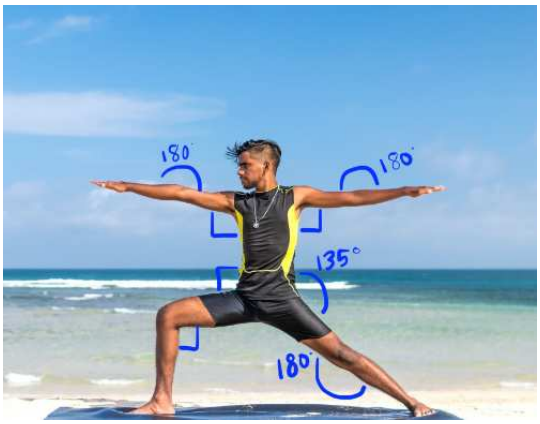


Figura 22: Posizione di yoga del Guerriero II (Esempio tratto dal sito di ML Kit)

Questa posa può essere descritta come la seguente combinazione di angoli delle parti del corpo approssimativi:

- Angolo di 90 gradi su entrambe le spalle
- 180 gradi su entrambi i gomiti

- Angolo di 90 gradi nella parte anteriore e in vita
- Angolo di 180 gradi alla schiena
- Angolo di 135 gradi in vita

Possiamo utilizzare i punti di riferimento per la posa per calcolare queste angolazioni. Ad esempio, l'angolo nella parte anteriore destra e in vita è l'angolo tra la linea dalla spalla destra all'anca destra e la linea dall'anca destra al ginocchio destro.

Una volta calcolati tutti gli angoli necessari per identificare la posa, possiamo controllare se c'è una corrispondenza, nel qual caso abbiamo riconosciuto la posa.

Lo snippet di codice riportato di seguito illustra come utilizzare le coordinate X e Y per calcolare l'angolo tra due parti del corpo. Questo approccio alla classificazione ha alcune limitazioni. Selezionando solo X e Y, gli angoli calcolati variano in base all'angolo tra il soggetto e la fotocamera. Otterremo i migliori risultati con un'immagine frontale, dritta e senza ostacoli. Si può anche provare a estendere questo algoritmo utilizzando la coordinata Z per vedere se ha un rendimento migliore per altri casi d'uso.

Calcolo degli angoli di riferimento su Android

Il metodo seguente calcola l'angolo tra tre punti di riferimento. Garantisce che l'angolo restituito sia compreso tra 0 e 180 gradi.

```
fun getAngle(firstPoint: PoseLandmark, midPoint: PoseLandmark, lastPoint:
    PoseLandmark): Double {
        var result = Math.toDegrees(atan2(lastPoint.getPosition().y -
    midPoint.getPosition().y,
                lastPoint.getPosition().x - midPoint.getPosition().x)
                - atan2(firstPoint.getPosition().y - midPoint.getPosition().
    y,
                firstPoint.getPosition().x - midPoint.getPosition().x))
        result = Math.abs(result) // Angle should never be negative
        if (result > 180) {
            result = 360.0 - result // Always get the acute representation
    of the angle
        }
        return result
    }
```

Ecco come calcolare l'angolazione sull'anca destra:

```
val rightHipAngle = getAngle(
                pose.getPoseLandmark(PoseLandmark.Type.RIGHT_SHOULDER),
                pose.getPoseLandmark(PoseLandmark.Type.RIGHT_HIP),
                pose.getPoseLandmark(PoseLandmark.Type.RIGHT_KNEE))
```

4.4 Google Colab

Cos'è Colab?

Colab, o "Colaboratory" [6], permette di scrivere ed eseguire Python direttamente nel browser con

- Nessuna configurazione necessaria
- Accesso alle GPU senza costi

- Condivisione semplificate

Introduzione

Le pagine web che si aprono in Colab non sono pagine statiche, ma costituiscono un ambiente interattivo chiamato **blocco note Colab**, che permette di srivere ed eseguire codice.

Ad esempio, qui sotto vediamo uno screen di una **cella di codice** con un breve script Python che calcola un valore, lo archivia in una variabile e stampa i risultati:

Figura 23: Esempio di blocco note Colab.

Per eseguire il codice bisogna selezionare la cella con un click e poi premere il pulsante "Riproduci" a sinistra del codice. Per modificare il codice, fare click sulla cella e poi si può modificare. Le variabili che si definiscono in una cella possono essere usate in seguito in altre celle.

I blocchi note Colab permettono di combinare codice eseguibile e RTF in un unico documento, insieme a immagini, HTML, LaTeX e altro ancora. Quando si creano dei blocchi note Colab, questi vengono archiviati nell'account personale Google Drive. Si può condividere facilmente i blocchi note Colab con collaboratori o amici e dare loro la possibilità di commentare o modificare.

Data science

Con Colab si può sfruttare tutta la potenza delle librerie Python per analizzare e visualizzare i dati. La seguente cella di codice usa **numpy** per generare alcuni dati casuali e usa **matplotlib** per visualizzarli.

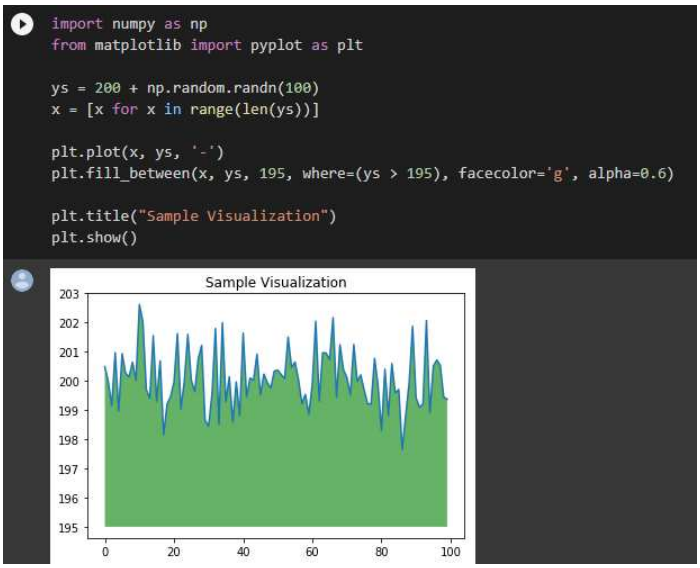


Figura 24: Esempio di utilizzo di Colab per la Data Science.

Si possono importare i dati nei blocchi note Colab dall'account Google Drive, inclusi i fogli di lavoro, da GitHub e molte altre fonti.

Machine learning

Con Colab si può importare un set di dati immagine, addestrare un classificatore di immagini e valutare il modello, il tutto in poche righe di codice [5]. I blocchi note Colab eseguono il codice sui server cloud di Google, il che significa che puoi utilizzare la potenza dell'hardware Google, tra cui GPU e TPU, a prescindere dalla potenza della tua macchina. Ti serve solo un browser.

Colab è ampiamente utilizzato dalla community del machine learning con applicazioni che includono:

- Introduzione a TensorFlow
- Sviluppo e addestramento di reti neurali
- Esperimenti con TPU
- Diffusione della ricerca sull'AI
- Creazione di tutorial

Pose Classification

Di seguito vengono riportati degli screen presi dalla pagina web di Colab per la creazione delle pose personalizzate.
Tengo a precisare che il file CSV importato nel progetto è stato creato utilizzando il codice Colab "Pose classification extended" [4], ovvero un codice più complesso per ridurre notevolmente le imprecisioni dell'addestramento. Tuttavia ho deciso di riportare la versione "basic" [3], in quanto più breve e più facile da comprendere, ma più soggetta ad errori ed imprecisioni nel riconoscimento della posa.

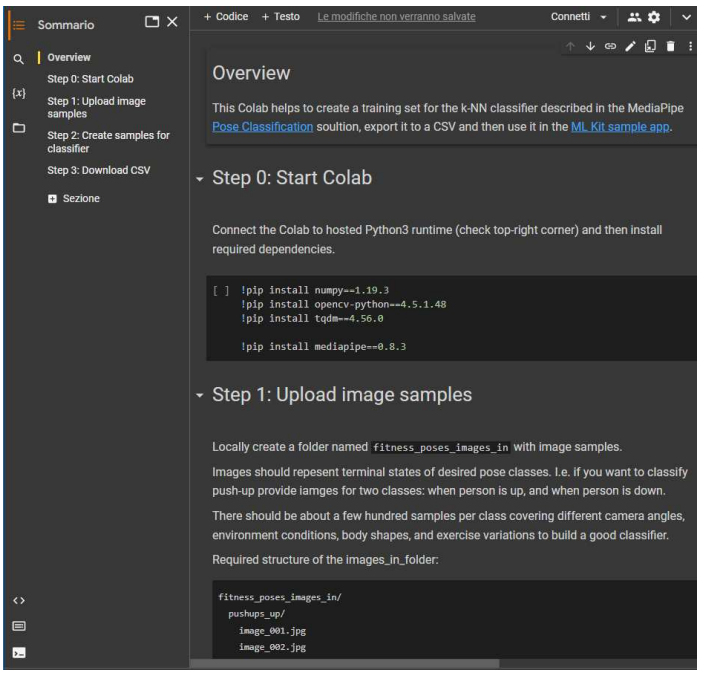


Figura 25: In questa parte di codice si fa partire Colab e si installano le librerie necessarie per la creazione della classificazione.

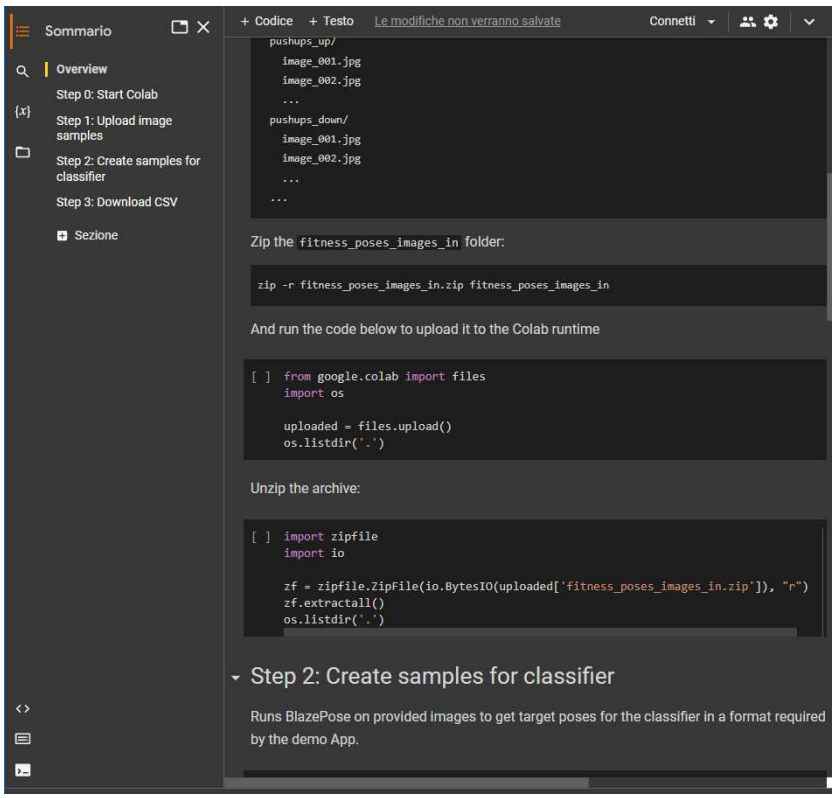


Figura 26: Una volta creata la cartella formattata come richiesto, la si comprime in un file .zip, ed infine viene caricata nel blocco Colab.

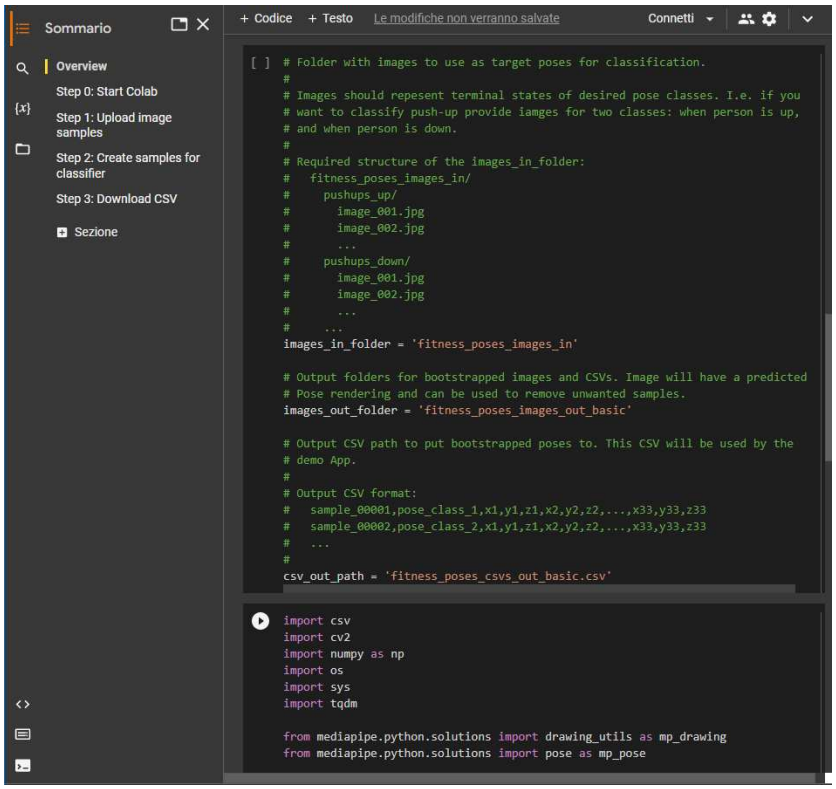


Figura 27: Successivamente vengono creati gli esempi di posa per poi iniziare a trasformarli nel formato CSV.

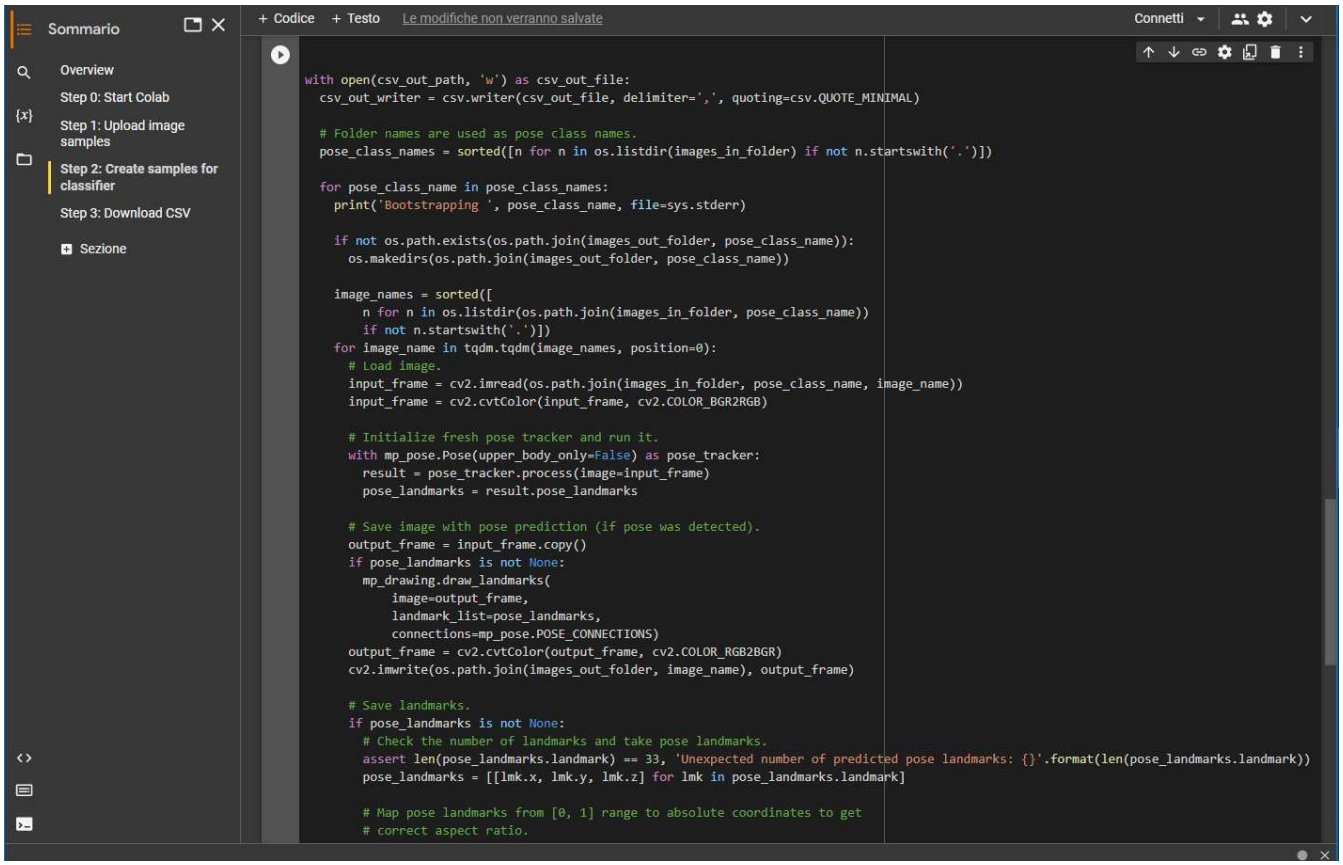


Figura 28: Le pose vengono mappate con i landmarks.

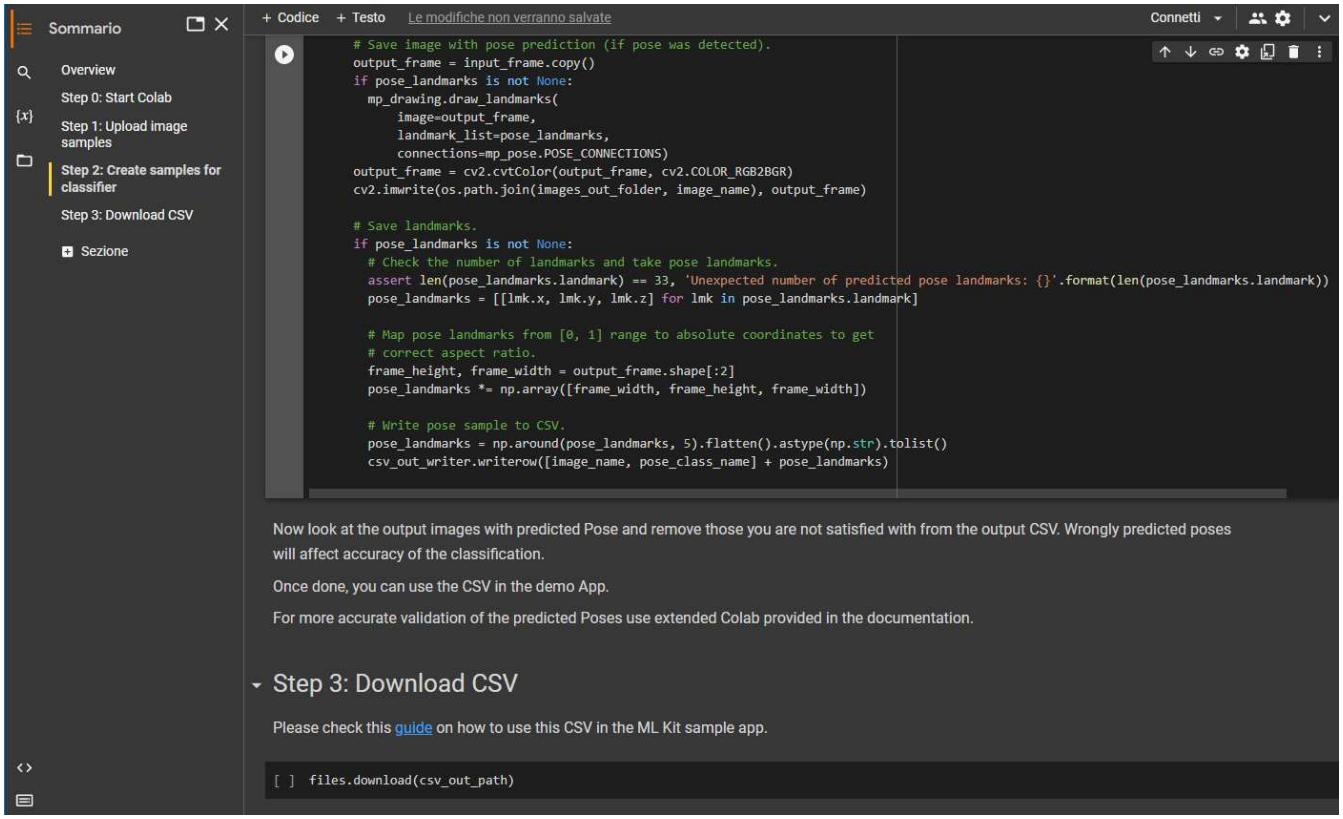


Figura 29: Infine viene scritta la posa in formato CSV ed è poi possibile effettuare il download del file.

4.5 BMI Detector/Estimator (ML Kit)

Una seconda funzionalità che ho voluto provare ad implementare all'interno dell'applicazione è la possibilità di riconoscere il livello di BMI di una persona tramite la fotocamera del cellulare grazie all'utilizzo delle API di etichettatura delle immagine di ML Kit [8].

Con le API di etichettatura delle immagini di ML Kit si può rilevare ed estrarre informazioni sulle entità in un'immagine in un ampio gruppo di categorie. Il modello predefinito di etichettatura delle immagini è in grado di identificare oggetti, luoghi, attività, specie di animali, prodotti e altro ancora.

Noi non utilizzeremo il modello predefinito, bensì utilizzeremo un nostro modello di classificazione delle immagini per riconoscere un essere umano, e di conseguenza la qualità della sua massa grassa. Utilizzeremo TensorFlow Lite per addestrare il nostro modello personalizzato.

Utilizzo di un modello TensorFlow Lite personalizzato

Il modello di etichettatura delle immagini di base di ML Kit è progettato per un uso generico. È addestrato a riconoscere 400 categorie che descrivono gli oggetti più comuni nelle foto. La nostra app ha bisogno di un modello di classificazione delle immagini specializzato che riconosca un numero più ristretto di categorie, ovvero l'essere umano e la sua corporatura.

Questa API consente di adattare il modello a un particolare caso d'uso supportando modelli personalizzati di classificazione di immagini da una vasta gamma di origini. I modelli personalizzati possono essere abbinati alla nostra app o scaricati in modo dinamico dal cloud utilizzando il servizio di deployment dei modelli di Firbase Machine Learning.

Etichetta le immagini con un modello personalizzato su Android

Esistono due modi per integrare le etichette delle immagini con i modelli personalizzati: raggruppando la pipeline come parte della app o utilizzando una pipeline non integrata che dipende da Google Play Services. Se si seleziona la pipeline in bundle, la nostra app sarà più piccola, ma per questione di praticità noi useremo una pipeline che sarà collegata in modo statico alla nostra app al momento della build.

Esistono due modi per integrare un modello personalizzato: raggruppare il modello inserendolo nella cartella degli asset dell'app o scaricarlo in modo dinamico da Firebase. Sempre per praticità verrà utilizzato il primo metodo.

0. Inserimento delle dipendenze all'interno dell'app

1. Nel file **build.gradle** a livello di progetto, mi assicuro di includere il repository Maven di Google nelle sezioni *buildscript* e *allprojects*.
2. Aggiungo le dipendenze per le librerie Android di ML Kit al file gradle a livello di app del modulo, ovvero **app/build.gradle**. Come abbiamo accennato prima, useremo la pipeline integrata direttamente nell'app, di conseguenza aggiungiamo la seguente dipendenza [4]:

```
dependencies {//...
  //Use this dependency to bundle the pipeline with your app
  implementation 'com.google.mlkit:image-labeling-custom:17.0.1'
}
```

[4]Per utilizzare le modalità in remoto che abbiamo accennato prima servirà aggiungere ulteriori dipendenze, che attualmente non sono riportate, in quanto non necessarie a noi per questo caso d'uso.

1. Caricamento del modello

Configurazione dell'origine di un modello locale[5]

Per raggruppare il modello con la nostra app:

1. Copiamo il file del modello (che di solito termina con **.tflite** o **.lite**) nella cartella ***assets/*** dell'app. Potrebbe essere necessario crearla (noi già l'abbiamo creata per la classificazione delle pose).

2. Quindi, aggiungiamo il seguente codice al file **build.gradle** dell'app per assicurarci che Gradle non comprima il file del modello durante la creazione dell'app:

```
android {
    // ...
    aaptOptions {
        noCompress "tflite"
        // or noCompress "lite"
    }
}
```

   Il file del modella verrà incluso nel pacchetto dell'app e sarà disponibile per il ML Kit come asset non elaborato.

   A partire dalla versione **4.1 del plug-in Android per Gradle**, verrà aggiunto per impostazione predefinita .tflite all'elenco noCompress e non è più necessario quanto sopra.

3. Creiamo l'oggetto **LocalModel**, specificando il percorso del file del modello:

```
val localModel = LocalModel.Builder()
        .setAssetFilePath("model.tflite")
        // or .setAbsoluteFilePath(absolute file path to model file)
        // or .setUri(URI to model file)
        .build()
```

Configurazione del labeler immagini

Dopo aver configurato l'origine del modello, creiamo un oggetto **ImageLabeler**.

Sono disponibili le seguenti opzioni:

- **confidenceThreshold:** punteggio di affidabilità minimo delle etichette rilevate. Se non viene impostato, verrà usata qualsiasi soglia di classificazione specificata dai metadati del modello. Se il modello non contiene metadati o i metadati non specificano una soglia di classificazione, verrà utilizzata una soglia predefinita di 0,0.

- **maxResultCount:** numero massimo di etichette da restituire. Se non viene impostato, verrà utilizzato il valore predefinito 10.

Nel nostro caso abbiamo un solo modello con bundle in locale, di conseguenza è sufficiente creare un etichettatore dall'oggetto **LocalModel**:

```
val customImageLabelerOptions = CustomImageLabelerOptions.Builder(localModel
    )
    .setConfidenceThreshold(0.5f)
    .setMaxResultCount(5)
    .build()
val labeler = ImageLabeling.getClient(customImageLabelerOptions)
```

[5] È possibile configurare, come detto precedentemente, un'origine modello ospitata da Firebase.

2. Prepariamo l'immagine di input

Quindi, per ogni immagine a cui vogliamo assegnare un'etichetta, creiamo un oggetto **InputImage**. Il labeler immagini funziona più velocemente quando utilizziamo un **Bitmap** o, se utilizziamo l'API **camera2** un **media.Image** YUV_420_888, che è consigliato quando possibile.

Possiamo creare un oggetto **InputImage** da origini diverse, le stesse che abbiamo visto nella sezione **4.2 Pose Detector (ML Kit)**, perciò eviterò di riportarle di seguito.

3. Eseguiamo l'etichettatore delle immagini

Per etichettare gli oggetti in un'immagine, passiamo l'oggetto **image** al metodo **process()**.

```
labeler.process(image)
        .addOnSuccessListener { labels ->
            // Task completed successfully
            // ...
        }
        .addOnFailureListener { e ->
            // Task failed with an exception
            // ...
        }
```

4. Ottenere informazioni sulle entità etichettate

Se l'operazione di etichettatura delle immagini ha esito positivo, viene passato un elenco di oggetti **ImageLabel** al listener di successo. Ogni oggetto **ImageLabel** rappresenta qualcosa che è stato etichettato nell'immagine. Possiamo ottenere la descrizione del testo di ogni etichetta (se disponibile nei metadati del file del modello TensorFlow Lite), il punteggio di affidabilità e l'indice. Ad esempio:

```
for (label in labels) {
    val text = label.text
    val confidence = label.confidence
    val index = label.index
}
```

4.6 Modelli personalizzati con ML Kit

Per impostazione predefinita, le API di ML Kit utilizzano modelli di machine learning addestrati da Google. Questi modelli sono progettati per coprire una vasta gamma di applicazioni. Tuttavia, alcuni casi d'uso richiedono modelli più mirati. Ecco perché alcune API di ML Kit ora consentono di sostituire i modelli predefiniti con modelli TensorFlow Lite personalizzati.

Sia l'etichettatura delle immagini che l'API Object Detection & Tracking supportano i modelli di classificazione delle immagini personalizzati. Sono compatibili con la selezione di modelli preaddestrati di alta qualità su TensorFlow Hub o con un modello nostro personalizzato addestrato con TensorFlow, AutoML Vision Edge o TensorFlow Lite Model Maker.

Vantaggi dell'utilizzo del kit ML con i modelli personalizzati

I vantaggi dell'utilizzo di un modello di classificazione delle immagini personalizzato con ML Kit sono:

- **API di alto livello facili da utilizzare:** non bisogna preoccuparsi di input/output del modello d ibasso livello, di gestire la pre-/elaborazione post-immagine o di creare una pipeline di elaborazione.
- **Non bisogna preoccuparsi di mappare le etichette autonomamente**, ML Kit estrae le etichette dai metadati dei modelli TFLite ed esegue la mappatura al posto nostro.
- **Supporta modelli personalizzati da un'ampia gamma di origini**, dai modelli pre-addestrati pubblicati su TensorFlow Hub a nuovi modelli addestrati con TensorFlow, AutoML Vision Edge o TensorFlow Lite Model Maker.
- **Supporta i modelli ospitati con Firebase**. Riduce le dimensioni dell'APK scaricando i modelli on demand. Esegui il push degli aggiornamenti del modello senza ripubblicare l'app ed eseguire test A/B con Firebase Remote Config.
- Ottimizzata per l'integrazione con le **API Fotocamera di Android**.

In particolare, per Rilevamento e monitoraggio degli oggetti:

- **Migliora la precisione della classificazione** individuando prima gli oggetti ed esegui solo il classificatore nell'area dell'immagine correlata.
- **Esperienza interattiva in tempo reale** fornendo agli utenti un feedback immediato sugli oggetti quando vengono rilevati e classificati.

Utilizzo di un modello di classificazione di immagini preaddestrato

TensorFlow Hub [15] offre una vasta gamma di modelli di classificazione di immagini preaddestrati, di vari autori di modelli, che possono essere utilizzati com le API Image Labeling e Object Detection & Tracking. Per usare questo metodo dovremmo:

1. Scegliere un modello dalla raccolta di modelli compatibili con ML Kit.
2. Scaricare il file modello .tflite dalla pagina dei dettagli del modello. Se disponibile, scegliere un formato del modello con metadati.
3. Seguire le linee guida descritte nella sezione precedente per raggruppare il file del modello con il nostro progetto e utilizzarlo nella nostra applicazione Android.

Addestrare un modello di classificazione delle immagini

Se nessun modello di classificazione delle immagine preaddestrato si adatta alle nostre esigenze, esistono diversi modi per addestrare il nostro modello TensorFlow Lite:

- **AutoML Vision Edge**
  - Offerta tramite AI Google Cloud
  - Crea modelli di classificazione delle immagini all'avanguardia
  - Valuta facilmente il rendimento e le dimensioni
- **Modello di modello TensorFlow Lite**
  - Addestra nuovamente un modello (trasferimento dell'apprendimento), richiede meno tempo e richiede meno dati rispetto all'addestramento di un modello da zero

- **Convertire un modello TensorFlow in TensorFlow Lite**
  - Addestra un modello con TensorFlow e convertilo in TensorFlow Lite

AutoML Vision Edge

I modelli di classificazione delle immagini addestrati utilizzando **AutoML Vision Edge** sono supportati dai modelli personalizzati nelle API Image Labeling e API Detection e Tracking. Queste API supportano anche il download di modelli ospitanti con il deployment di modelli Firebase.

ML Kit supporta solo modelli di classificazione delle immagini personalizzati. Sebbene AutoML Vision consenta l'addestramento di modelli di rilevamento di oggetti, non possono essere utilizzati con ML Kit (per questo non verrà utilizzato per il nostro progetto).

Creazione modelli TensorFlow Lite

La libreria TFLite Model Maker semplifica il processo di adattamento e conversione di un modello di rete neurale TensorFlow in particolari dati di input al momento del deployment di questo modello per le applicazioni ML on-device. Possiamo seguire la classificazione di Colab per le immagini con TensorFlow Lite Model Maker [2].

Modelli creati utilizzando il programma di conversione TensorFlow Lite

Nel caso si abbia già un modello di classificazione delle immagini TensorFlow, possiamo convertirlo utilizzando il convertitore TensorFlow Lite [14], assicurandoci, però, che il modello creato soddisfi i requisiti di compatibilità riportati nella sottosezione successiva.

Compatibilità dei modelli TensorFlow Lite

Possiamo utilizzare qualsiasi modello di classificazione delle immagini TensorFlow Lite preaddestrato, a condizione che soddisfi i seguenti requisiti:

Tensor

- Il modello deve avere un solo tensore di input con i seguenti vincoli:
  - I dati sono in formato pixel RGB.
  - I dati sono di tipo UNIT8 o FLOAT32. Se il tipo di tensore di input è FLOAT32, deve specificare le opzioni di normalizzazione collegando i **metadati**.
  - Il tensore ha 4 dimensioni: BxHxLxC, dove:
    - **B** è la dimensione del batch. Deve essere 1 (l'inferenza su batch più grandi non è supportata).
    - **W** e **H** sono la larghezza e l'altezza di input.
    - **C** è il numero di canali previsti. Deve essere 3.
- Il modello deve avere almeno un tensore di output con N classi e due o quattro dimensioni:
  - (1 x N)
  - (1 x 1 x 1 x N)

Metadati
Si possono aggiungere metadati al file TensorFlow Lite [13].

Per utilizzare un modello con tensore di input FLOAT32, bisogna specificare le **NormalizationOptions** nei metadati[6].

Inoltre, è anche consigliato di collegare i seguenti metadati al tensore di output **TensorMetadata**:

- Una mappa delle etichette che specifica il nome di ciascuna classe di output, come AssociatedFile con tipo TENSOR_AXIS_LABELS (altrimenti possono essere restituiti solo gli indici delle classi di output);
- Una soglia di punteggio predefinita al di sotto della quale i risultati sono considerati troppo bassi per essere restituiti, come ProcessUnit con ScoreThresholdingOptions;

Image classification with TensorFlow Lite Model Maker

Per la creazione del modello abbiamo usufruito del codice di TensorFlow Lite Model Maker [2] ospitato su Google Colab. Di seguito sono riportati gli screen dei blocchi di codice con didascalia annessa. Nella sezione "Considerazioni" parlerò del dataset utilizzato e delle sue limitazioni.

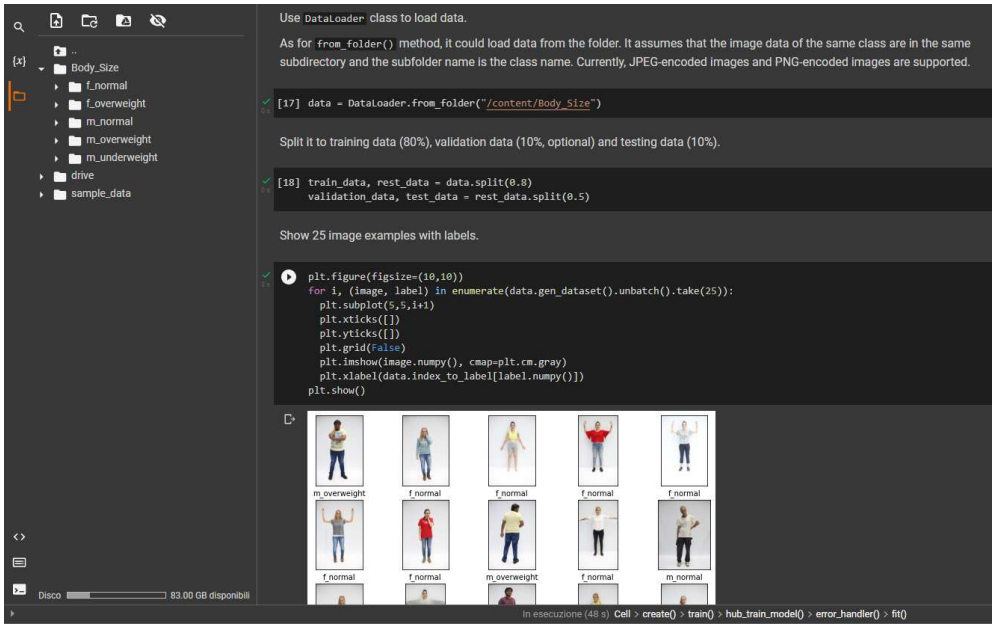


Figura 30: Tramite il metodo "DataLoader.from_folder" vado a specificare il percorso della cartella "Body_Size" (caricata su Colab) contenente le immagini per il Training del modello suddivise per categorie: dove "f" sta per **female** e "m" sta per **male**. Da notare che dopo il blocco [17] vi è uno split randomico del dataset in tre parti: train_data (80% del dataset), validation_data (10% del dataset) e test_data (10% del dataset). In modo da poter valutare e testare il modello su immagini mai viste prima.

[6]Link: **shorturl.at/buyNV**

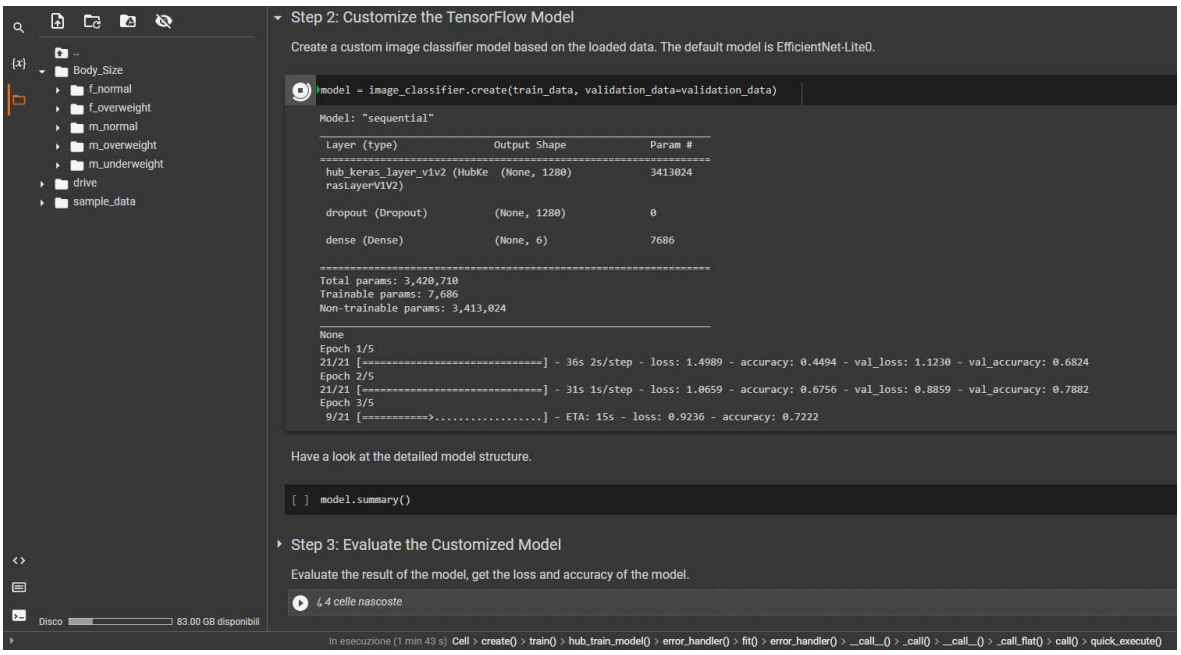


Figura 31: Tramite il metodo "image_classifier.create" e passandogli le variabili del train_data e del validation_data, andremo a creare il nostro **model**.

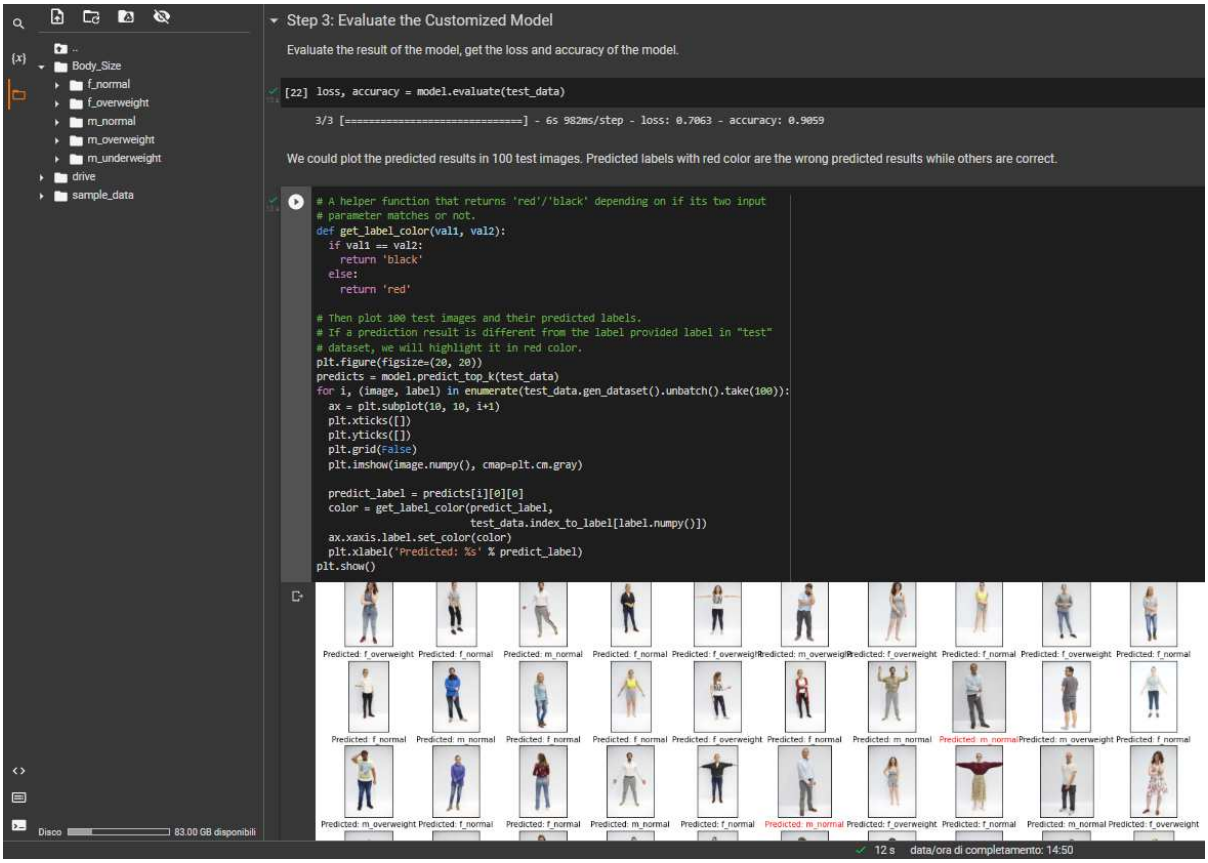


Figura 32: Tramite il metodo "model.evaluate" è possibile ottenere una stima dell'accuratezza del modello. Successivamente, con il blocco immediatamente dopo, è possibile effettuare il test sul test_data creato in precedenza. I label di colore rosso sono le previsioni della classificazione sbagliate, mentre quelle di colore nero sono quelle corrette.

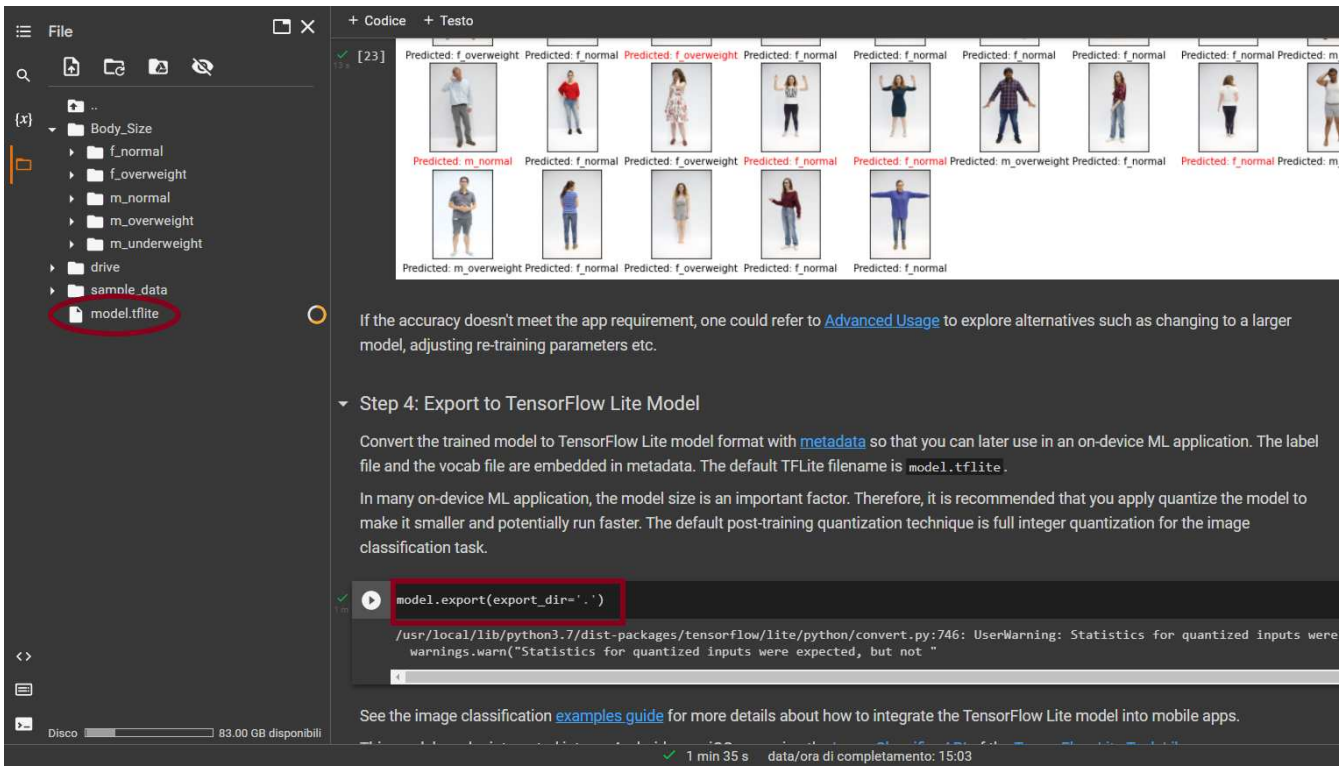


Figura 33: Infine, tramite il metodo "export" possiamo esportare il nostro modello. Il modello avrà estensione **.tflite** ed è già pronto per essere inserito all'interno della cartella "assets" nel nostro progetto.

Considerazioni sul dataset utilizzato

Il DataSet utilizzato per il training è stato messo a disposizione dal sito "SHAPY" [12]. Le immagini sono state classificate secondo il sesso del soggetto e la sua corporatura. Un primo problema sorto è che il modello non è ottimizzato al massimo in quanto la quantità di immagini a disposizione era limitata ed imprecisa. Inoltre, nonostante la suddivisione della corporatura, ancora c'è difficoltà nel riconoscimento della stazza del soggetto attraverso la fotocamera, in quanto i modelli utilizzati per il training non presentano una evidente differenza di taglia fisica tra normal e over/underweight. Questo problema è accentuato dal fatto che i vestiti indossati dai modelli possono causare confusione al modello.

Con l'utilizzo di un DataSet per il training più preciso, efficiente e completo sicuramente il modello risulterebbe più ottimizzato di quello utilizzato.

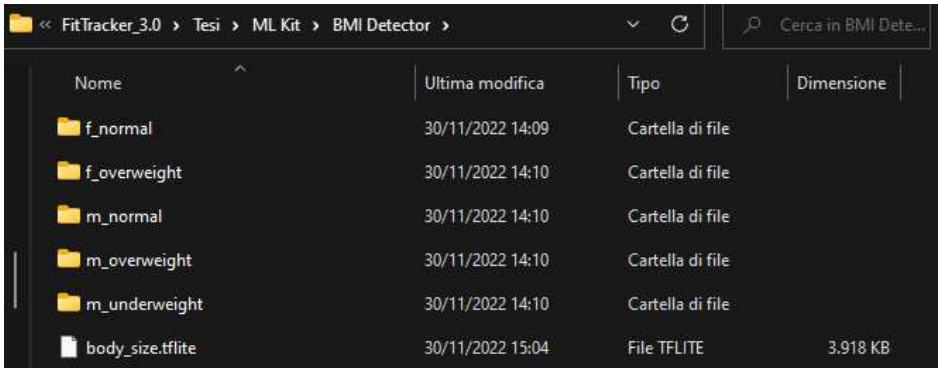


Figura 34: Le immagini prese dal sito sono state visionate e classificate secondo queste 5 categorie. Successivamente questa cartella è stata caricata su Colab come DataSet e utilizzato per la creazione del modello (**body_size.tflite**)

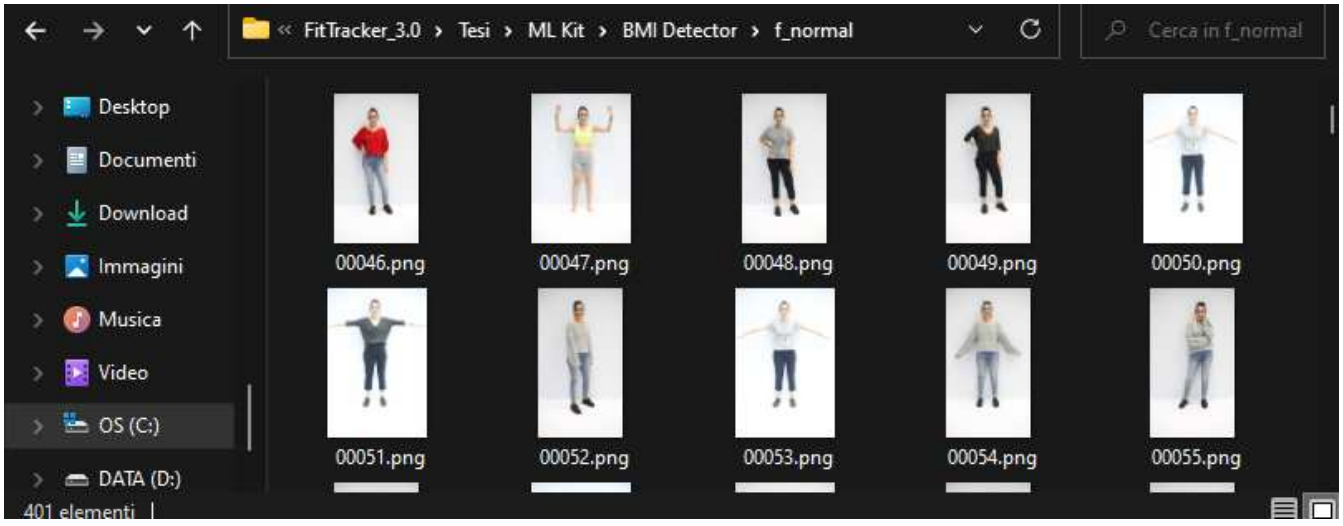


Figura 35: Prendiamo come esempio la cartella f_normal, contiene 401 immagini di ragazze in posa diverse e con abbigliamenti differenti.

LabelDetectorProcessor Class

```
package com.example.fittracker.funzioni.labeldetector

import android.content.Context
import android.util.Log
import com.example.fittracker.funzioni.posedetector.GraphicOverlay
import com.example.fittracker.funzioni.posedetector.kotlin.
    VisionProcessorBase
import com.google.android.gms.tasks.Task
import com.google.mlkit.vision.common.InputImage
import com.google.mlkit.vision.label.ImageLabel
import com.google.mlkit.vision.label.ImageLabeler
import com.google.mlkit.vision.label.ImageLabelerOptionsBase
import com.google.mlkit.vision.label.ImageLabeling
import java.io.IOException

/** Custom InputImage Classifier*/
class LabelDetectorProcessor(context: Context, options:
    ImageLabelerOptionsBase) :
  VisionProcessorBase<List<ImageLabel>>(context) {

  private val imageLabeler: ImageLabeler = ImageLabeling.getClient(options)

  override fun stop() {
    super.stop()
    try {
      imageLabeler.close()
    } catch (e: IOException) {
      Log.e(
        TAG,
        "Exception thrown while trying to close ImageLabelerClient: $e"
      )
    }
  }

  override fun detectInImage(image: InputImage): Task<List<ImageLabel>> {
    return imageLabeler.process(image)
  }

  override fun onSuccess(labels: List<ImageLabel>, graphicOverlay:
     GraphicOverlay) {
    graphicOverlay.add(LabelGraphic(graphicOverlay, labels))
    logExtrasForTesting(labels)
  }

  override fun onFailure(e: Exception) {
    Log.w(TAG, "Label detection failed.$e")
  }

  companion object {
    private const val TAG = "LabelDetectorProcessor"

    private fun logExtrasForTesting(labels: List<ImageLabel>?) {
      if (labels == null) {
        val MANUAL_TESTING_LOG = null
        Log.v(MANUAL_TESTING_LOG, "No labels detected")
      } else {
        for (label in labels) {
          val MANUAL_TESTING_LOG = null
```

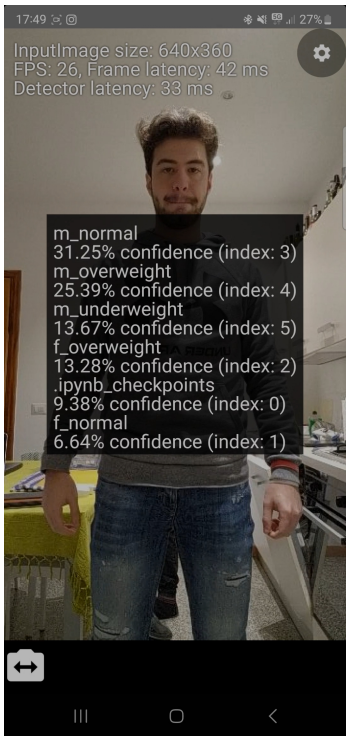
```
          Log.v(
            MANUAL_TESTING_LOG,
            String.format("Label %s, confidence %f", label.text, label.
    confidence)
          )
        }
      }
    }
  }
}
```

LabelGraphic Class

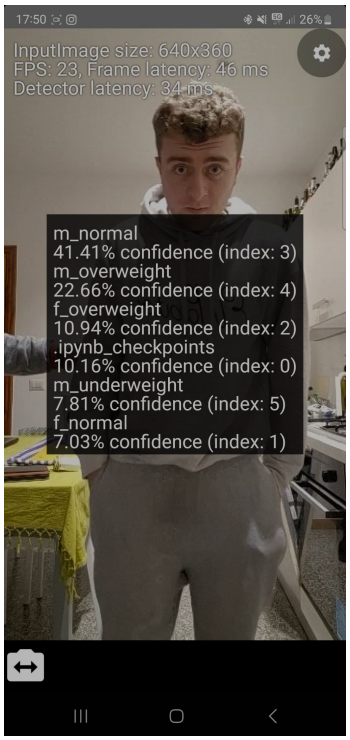
```
package com.example.fittracker.funzioni.labeldetector

import android.graphics.Canvas
import android.graphics.Color
import android.graphics.Paint
import com.example.fittracker.funzioni.posedetector.GraphicOverlay
import com.google.mlkit.vision.label.ImageLabel
import java.util.Locale

/** Graphic instance for rendering a label within an associated graphic
    overlay view.  */
class LabelGraphic(
  private val overlay: GraphicOverlay,
  private val labels: List<ImageLabel>) : GraphicOverlay.Graphic(overlay) {
  private val textPaint: Paint = Paint()
  private val labelPaint: Paint

  init {
    textPaint.color = Color.WHITE
    textPaint.textSize = TEXT_SIZE
    labelPaint = Paint()
    labelPaint.color = Color.BLACK
    labelPaint.style = Paint.Style.FILL
    labelPaint.alpha = 200
  }

  @Synchronized
  override fun draw(canvas: Canvas) {
    // First try to find maxWidth and totalHeight in order to draw to the
    center of the screen.
    var maxWidth = 0f
    val totalHeight = labels.size * 2 * TEXT_SIZE
    for (label in labels) {
      val line1Width = textPaint.measureText(label.text)
      val line2Width =
        textPaint.measureText(
          String.format(
            Locale.US,
            LABEL_FORMAT,
            label.confidence * 100,
            label.index
          )
        )

      maxWidth = maxWidth.coerceAtLeast(line1Width.coerceAtLeast(line2Width)
    )
    }
```

```
    val x = 0f.coerceAtLeast(overlay.width / 2.0f - maxWidth / 2.0f)
    var y = 200f.coerceAtLeast(overlay.height / 2.0f - totalHeight / 2.0f)

    if (labels.isNotEmpty()) {
      val padding = 20f
      canvas.drawRect(
        x - padding,
        y - padding,
        x + maxWidth + padding,
        y + totalHeight + padding,
        labelPaint
      )
    }

    for (label in labels) {
      if (y + TEXT_SIZE * 2 > overlay.height) {
        break
      }
      canvas.drawText(label.text, x, y + TEXT_SIZE, textPaint)
      y += TEXT_SIZE
      canvas.drawText(
        String.format(
          Locale.US,
          LABEL_FORMAT,
          label.confidence * 100,
          label.index
        ),
        x, y + TEXT_SIZE, textPaint
      )
      y += TEXT_SIZE
    }
  }

  companion object {
    private const val TEXT_SIZE = 60.0f
    private const val LABEL_FORMAT = "%.2f%% confidence (index: %d)"
  }
}
```

Screenshot all'interno dell'applicazione



Riconoscimento di un Maschio con corporatura Normale con una confidenza del 31.25%



Riconoscimento di un Maschio con corporatura Normale con una confidenza del 41.41%

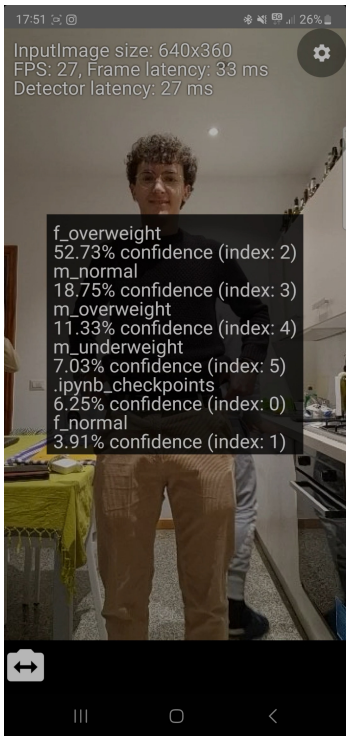


Figura 36: Riconoscimento di una Femmina con corporatura Sovrappeso con una confidenza del 52.73%. Questo è chiaramente un errore dovuto al dataset utilizzato per il modello, in quanto il soggetto preso in riferimento è di sesso maschile.

4.7 Conclusioni e miglioramenti futuri

Prendendo in considerazione le funzioni aggiunte possiamo dire che in generale funzionano tutte correttamente e non si sono riscontrati bug riguardanti l'implementazione all'interno dell'applicazione.

Miglioramenti futuri potrebbero riguardare:

Alan:

- Aggiunta di nuovi comandi vocali come l'avvio di un cronometro o di un timer;
- Aggiunta di risposte a domande che riguardano il fitness, muscoli e l'importanza di una buona alimentazione variegata.

Pose Detector:

- Aggiunta di nuovi esercizi da riconoscere e contare;
- Possibilità di utilizzare l'applicazione in palestra per vedere il corretto svolgimento di un determinato esercizio;
- Miglioramento dell'overlay e interazione con utente.

BMI Detector:

- Miglioramento del modello con l'utilizzo di un dataset più completo;
- Miglioramento del layout dei label all'interno dell'applicazione;
- Rilevamento del BMI tramite fotografia statica e non solo in fotocamera live;
- Previsione del BMI tramite rilevamento facciale;
- Miglioramento dell'overlay e interazione con utente.

Sitografia

[1] Google Developers - Android. *Welcome to Android Developers*. URL: https://developer.android.com/.

[2] Google Colab. *Model Maker Image Classification Tutorial*. URL: https://colab.research.google.com/github/tensorflow/tensorflow/blob/master/tensorflow/lite/g3doc/models/modify/model_maker/image_classification.ipynb#scrollTo=TUfAcER1oUS6.

[3] Google Colab. *Pose Classification (basic)*. URL: https://colab.research.google.com/drive/1z4IM8kG6ipHN6keadjD-F6vMiIIgViKK.

[4] Google Colab. *Pose Classification (extended)*. URL: https://colab.research.google.com/drive/19txHpN8exWhstO6WVkfmYYVC6uug_oVR.

[5] Google Colab. *TensorFlow 2 quickstart for beginners*. URL: https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb#scrollTo=3wF5wszaj97Y.

[6] Google Colab. *Un benvenuto a Colaboratory*. URL: https://colab.research.google.com/.

[7] ALAN AI inc. *Create a great User Experience for your App with onversational AI*. URL: https://alan.app/.

[8] Google Firebase - ML Kit. *Etichettatura delle immagini*. URL: https://developers.google.com/ml-kit/vision/image-labeling.

[9] Google Firebase - ML Kit. *Kit ML*. URL: https://developers.google.com/ml-kit/guides.

[10] Google Firebase - ML Kit. *Rilevamento della posa*. URL: https://developers.google.com/ml-kit/vision/pose-detection.

[11] Google - MediaPipe. *Pose Classification*. URL: https://google.github.io/mediapipe/solutions/pose_classification.

[12] SHAPY. *Human Bodies in the Wild*. URL: https://shapy.is.tue.mpg.de/datasets.html.

[13] TensorFlow. *Aggiunta di metadati ai modelli TensorFlow Lite*. URL: https://www.tensorflow.org/lite/models/convert/metadata.

[14] TensorFlow. *Panoramica della conversione del modello*. URL: https://www.tensorflow.org/lite/models/convert.

[15] TensorFlow. *TensorFlow Hub è un repository di modelli di machine learning addestrati*. URL: https://www.tensorflow.org/hub.

Ringraziamenti

Ora che ho completato questa tesi, ho finalmente l'occasione di ringraziare tutte le persone che mi hanno sostenuto durante questo viaggio accademico (e psicologico).

Innanzitutto, vorrei ringraziare il mio relatore, il Prof. Fiori Simone, che ha avuto la pazienza di leggere le mie innumerevoli bozze, di rispondere alle mie mille domande e di non ridere quando ho fatto domande sciocche. Grazie per avermi fatto sentire meno stupido del solito.

Inoltre, ringrazio il mio correlatore, il Prof. Storti Emanuele, per il suo supporto costante, i suoi preziosi consigli e la sua gentilezza.

Ringrazio anche i miei genitori e mia sorella, Andrea, Annalisa e Gemma, che mi hanno sempre sostenuto e incoraggiato, anche quando ho deciso di studiare argomenti che non sapevano neanche come si scrivessero. Grazie per aver creduto in me e per avermi dato la forza di portare avanti questo percorso. Un pensiero speciale va anche ai miei nonni: Rosanna, Giovanna e Dante, che mi hanno ispirato con la loro saggezza e il loro coraggio, e che mi hanno dato l'esempio di come affrontare le sfide della vita.

Un grazie speciale va anche a tutti i miei amici, in particolare Federico, Gregorio, Nicolò e Letizia, per le risate, gli scherzi, le giornate passate insieme ed i progetti di gruppo finiti all'ultimo momento. Grazie per avermi fornito la dose necessaria di sarcasmo e ironia quando ne avevo bisogno, e per avermi aiutato a sopravvivere alle notti insonni e alle caffetterie scadenti.

Infine, un ringraziamento speciale va a tutte le persone che ho incontrato durante il mio percorso, in particolare ai miei colleghi universitari e coinquilini. Grazie per aver reso le lezioni e le giornate ad Ancona divertenti e interessanti, e per avermi dato l'opportunità di incontrare persone fantastiche come voi. Siete stati il mio supporto morale quando l'università sembrava impossibile, e ho apprezzato ogni momento passato insieme, dalle serate universitarie ai pranzi a base di sushi.

Un consiglio per chi sta iniziando il percorso accademico: cercate di trovare compagni di corso che abbiano un senso dell'umorismo simile al vostro, altrimenti rischiate di impazzire! Grazie di nuovo a tutti voi, e non vedo l'ora di vedere le grandi cose che realizzerete in futuro.

SCAN ME