Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica Magistrale

FISHMETER: l'app per una scelta ecosostenibile

Tesi di Laurea in Laboratorio di Applicazioni Mobili

Relatore:
Chiar.mo Prof.
Luciano Bononi

Presentata da:
Lorenzo Vinci

Sessione 3
2015/2016

Dedicato a tutte le persone che mi sono rimaste vicine
e mi hanno sostenuto nonostante le difficoltà, il mio brutto
carattere ed i periodi bui:
Mauro: il mio unico e vero punto di riferimento;
Marco e Luca La Sala: veri amici con cui posso condividere tutto;
Claudia (amica dolce e sempre presente), Luca Pompei e Federico:
fedeli compagni di sventure informatiche senza il cui aiuto e
supporto non ce l'avrei fatta;
A Sara e Tatiana, perchè sanno sempre vedere il bello che c'è in me;
Ai miei genitori e mio fratello che, nonostante gli scontri continui,
mi ricordano la fortuna dell'avere una famiglia affettuosa;
Ed infine, alla mia cagnolina Puppy perchè rientrare a casa ogni
giorno e sentirsi amato e festeggiato dalla sua codina frenetica è
una gioia per il cuore e cibo per l'anima.
Grazie ...

Introduzione

Lo scopo di questo lavoro è stato quello di realizzare un'app, in collaborazione con l'Università degli Studi di Roma Tor Vergata, che fosse di supporto nello stabilire l'ecostenibilità del pesce comprato da potenziali acquirenti. In modo particolare, per ecosostenibilità dell'acquisto del pesce intendiamo principalmente due fattori:

- lunghezza minima del pesce pescato;
- attenzione posta sul pescare ed acquistare pesce nel giusto periodo dell'anno.

Col primo aspetto, intendiamo porre l'attenzione sul fatto che ogni esemplare di pesce deve essere di una certa lunghezza minima per essere pescato e poi messo in vendita mentre col secondo fattore intendiamo l'evitamento della pesca di certe specie di pesce qualora si trovino nella loro stagione riproduttiva. Pertanto, compito fondamentale dell'app presentata in questa tesi è quello di stimare la lunghezza di un pesce acquistato mediante una fotografia scattata allo stesso tramite uno *smartphone* e di verificare se esso sia stato venduto nella giusta stagione, preoccupandosi poi non solo di informare di conseguenza l'utente ma anche di salvare ed inviare una segnalazione riguardo l'esito dell'operazione a seguito di un'attenta raccolta di dati. Vedremo nel corso di questo documento quali siano stati tutti i passaggi di sviluppo di questa app e quali aspetti abbiano richiesto una maggiore attenzione per mantenere sia una semplicità d'uso nei confronti dell'utente sia un'implementazione rapida ma efficiente. In modo particolare questo lavoro è strutturato nel seguente modo:

Capitolo 1 : Stato dell'Arte

Nel primo capitolo verrà documentata la scena attuale in cui si è svolto lo sviluppo di questo progetto il che consiste essenzialmente nell'approfondire due aspetti:

1. la sostenibilità dei consumi di pesce, ossia come sia importante orientare il consumatore a scelte consapevoli e sostenibili di pesce;
2. il sistema operativo iOS di Apple per i dispositivi iPhone nella sua versione attuale e per il quale si è deciso di sviluppare tale app.

Capitolo 2 : Sviluppo e programmazione dell'app

In questo capitolo verranno presentati tutti gli *steps* attraverso i quali si è passati per realizzare l'app in questione. In modo particolare, verranno mostrare le difficoltà incontrare durante il processo e le soluzioni e scelte implementative che sono via via effettuate affinché il prodotto finale risultasse sia semplice nell'utilizzo sia efficacie e preciso per lo scopo che si propone.

Capitolo 3 : Conclusioni e sviluppi futuri

Infine, nell'ultimo capitolo verranno presnetati alcuni risultati ottenuti testanto l'applicazione su alcuni utenti e verificando che essa risponda a requisiti soddisfacenti di usabilità. Inoltre, verranno elencate alcune idee per possibili sviluppi futuri affinché l'app possa lavorare ancora più indipendente dall'utente ed in modo più efficace attraverso la sperimentazione di un particolare *framework* dedicato al settore della *computer vision*.

Indice

Elenco delle figure

Elenco dei frammenti di codice

Capitolo 1

Stato dell'arte

Prima di procedere all'illustrazione dello sviluppo del'applicazione, vogliamo analizzare da vicino non solo quale sia l'ambiente e lo scopo per i quali è stato proposto questo progetto, ma anche l'ambiente di sviluppo che si è scelto per realizzare l'app e come esso si presenta al giorno d'oggi a tutti gli sviluppatori che intendono avvicinarsi alla programmazione in ambiente Apple, di cui considereremo gli strumenti messi a disposizione per creare *software* per dispositivi mobili, in modo particolare per iPhone, ai quali questa app è rivolta.

1.1 Perchè creare un'app per raccogliere dati sull'ecosostenibilità?

L'idea di quest'app nasce dalla collaborazione tra il Dipartimento di Biologia Evoluzionistica della facoltà di Scienze dell'Università di Tor Vergata (Roma) e tra il Dipartimento di Informatica della Scuola di Scienze dell'Università di Bologna. Tale progetto comprende, in un'ottica moderna ed innovativa dei rapporti tra scienza e persone, la collaborazione dei cittadini nella raccolta di dati scientifici da usare nella ricerca. Questa metodologia è detta **Citizen Science** la quale, sebbene sia nata in America, è tuttora in continua espansione in molti paesi europei; tutti i suoi progetti sono basati sul seguente schema:

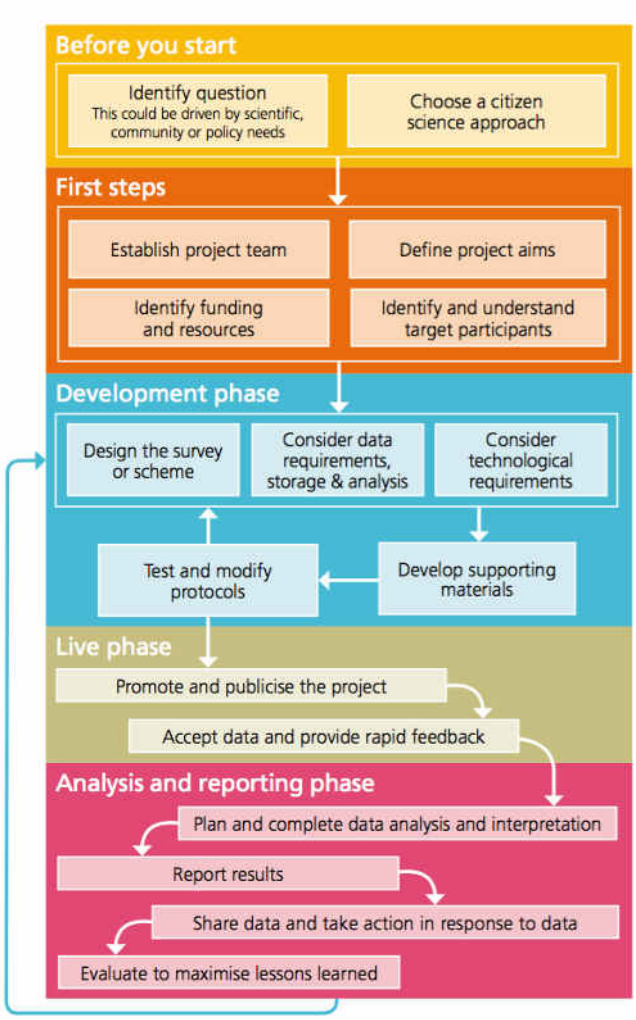


Figura 1.1: Svolgimento di un progetto di Citizen Science

da cui possiamo notare fin da subito che lo sviluppo di un'app per sostenere questo progetto fa parte della fase in azzurro indcata nell'immagine precedente. Dunque, ci si propone di raccogliere dei dati riguardo la sostenibiità degli acquisti di pesce dei cittadini nelle località di Orbetello e Monte Argentario, zone di residenza della Dott.ssa Flavia Bartoccioni dell'Università di Tor Vergata, la quale è la committente di questo lavoro in quanto aderente alla *Citizen Science*. Si vuole quindi misurare la sostenibilità degli acquisti ittici in quanto, come spiegheremo nella sezione seguente, si sta verificando un notevole calo della biodiversità negli

ambienti marini negli ultimi dieci anni: infatti, molte specie di pesce si trovano in uno stato di sovra-sfruttamento unito a numerosi problemi riguardo la pesca legale. Dunque, tra le diverse azioni che si possono prendere per limitare l'impatto ambientale troviamo l'**acquisto consapevole** che porti ad un consumo sostenibile. La *Citizen Science* è quindi un modo vero e proprio per fare ricerca alla cui raccolta dati viene in aiuto la cittadinanza. Ad esempio, un altro progetto di Citizen Science potrebbe essere la trattazione della diminuzione del lupo selvatico, valutandone l'estensione della popolazione nelle zone appenniniche. Per la raccolta dati si potrebbe, per esempio, chiedere ai cittadini di inviare delle segnalazioni ogni volta che vedono delle orme di lupo sui sentieri. Inoltre, questo metodo di ricerca non solo permette di raccogliere un numero di dati incredibile (che un ricercatore da solo non potrebbe mai ottenere) ma è anche fortemente basato sui moderni strumenti tecnologici che oggi sono alla portata ed al comune uso di tutti e quindi, non a caso, possiamo parlare sicuramente come esempio più lampante proprio degli smartphone e delle app. Oggi, l'importanza del supporto tecnologico per la Citizen Science è di vitale importanza. Si richiedeva quindi l'implementazione di un semplice software per smartphone che permettesse ai cittadini comuni l'inivio di dati scientifici in modo facile e veloce. Ultimo, ma non meno importante, quest'attività permette di fare anche **educazione ambientale** che mira ad una maggiore sensibilizzazione del cittadino. In merito a quest'ultimo punto, diamo adesso una breve panorama sulla situazione del mercato ittico in Italia.

1.2 La sostenibilità dei consumi di pesce

In Italia il fabbisogno dei prodotti ittici è in larga parte soddisfatto dalle importazioni. Secondo recenti dati, la propensione all'importo di pesce si aggira attorno al 74% mentre il saldo normalizzato, ossia il grado di dipendenza dall'estero di un paese in un determinato settore merceologico, è pari a circa -75% a testimonianza di un netto disavanzo della bilancia commerciale ittica e di uno scarso grado di copertura dell'*export*. Quanto minore è il saldo normalizzato, che (quando non espresso in percentuale) può assumere valori compresi tra – 1 (esportazioni nulle)

e 1 (importazioni nulle), più basso è livello di specializzazione esistente nel settore merceologico dato. Questo significa che in Italia la dipendenza dalle importazioni è costantemente aumentata nel corso degli ultimi anni e dunque la domanda interna viene soddisfatta soprattutto con un progressivo ricorso agli acquisti esteri le cui importazioni sono cresciute costantemente nel corso degli ultimi anni. Circa i tre quarti dell'importo in volume, dal 2010, è costituito da:

- numerosi prodotti lavorati e prevalentemente congelati che contengono molte varietà di molluschi e crostacei e pesci veri e propri interi o a filetti;
- grosse quantità di tonno per la relativa industria;
- pesci "di piccole dimensioni" che vengono preparati e conservati secondo varie modalità come il metterli in scatola sott'olio, affumicati oppure salati.

Per contro a quanto appena detto, la lista di importazione di prodotti freschi è molto più ridotta e si applica a poche specie (soprattutto orate, spigole e salmoni) e rappresentano circa il 45% dei prodotti freschi complessivamente importati.

Sulla base dello scenario appena illustrato, l'elevata dipendenza dalle importazioni, la progressiva contrazione del pescato nazionale unita alla concentrazione dei consumi su poche specie, il tutto viene rimandato al tema della sostenibilità in tavola. Secondo le ultime stime della FAO sugli *stock* ittici mondiali si ha che:

Percentuale di stock ittici	**Livello di sfruttamento**
28%	sfruttati in eccesso
3%	sfruttati completamente
1%	in fase di ricostituzione
53%	completamente sfruttati
3%	sotto-sfruttati
12%	moderatamente sfruttati

Tabella 1.1: Livello di sfruttamento degli *stock* ittici

Sebbene la maggior parte degli *stock* delle principali specie pescate a livello mondiali sono nella categoria dei completamente sfruttati (il che significa che le

catture di tali pesci sono vicine al massimo sostenibile senza registrare ulteriori aumenti), non bisogna comunque sottovalutare l'eccessivo sfruttamento di alcune risorse ittiche, che si trovano quindi ben oltre la capacità di rigenerazione dei relativi ecosistemi marini, come ad esempio il 35% degli *stock* di tonno. Tra queste occorre anche tenere in considerazione le catture accidentali di pesce non voluto, la pesca illegale ed i continui mutamenti climatici.

Dunque, in questo quadro relativo alla pesca mondiale, che per molti versi è preoccupante, la crescente domanda di pesce viene soddisfatta dall'incessante incremento dell'acquacoltura: infatti, stime relative allo scorso 2015 da poco conclusosi indicano che la quota di pesce destinato al consumo umano proveniente dall'acquacoltura sia di gran lunga superiore di quella proveniente dalla pesca. Tuttavia, lo sviluppo stesso dell'acquacoltura pone alcune questioni legate alla sua sostenibilità, considerando che:

- è richiesta una quantità di risorse notevole (si pensi ad esempio all'acquisizione di altri pesci per nutrire quelli allevati);
- è causato un enorme impatto ambientale a causa sia della distruzione ed inquinamento degli *habitat* naturali nonché dall'uso di enormi quantità di acqua;
- si deve porre particolare attenzione anche alla sicurezza alimentare a causa dell'uso negli allevamenti di antibiotici, farmaci ed altre sostanze chimice.

In questo contesto, con riferimento specifico alla realtà italiana, andrebbe valutata la necessità di ridurre i consumi o, ancora meglio, scegliere il pesce in maniera consapevole cercando di preferire le specie poco consumate (perchè poco conosciute ai consumatori stessi) che non sono né più care né meno buone di quelle comuni, ma sono lontane dalle abitudini aliminetari ormai sedimentate su poche specie. Ed ecco che a questo punto entra in gioco la questione del consumo sostenibile, il quale mira a contrastare il deterioramento dell'ambiente globale a causa degli insostenibili *patterns* di produzione e consumo dei paesi industrializzati. Lo possiamo, quindi, vedere come:

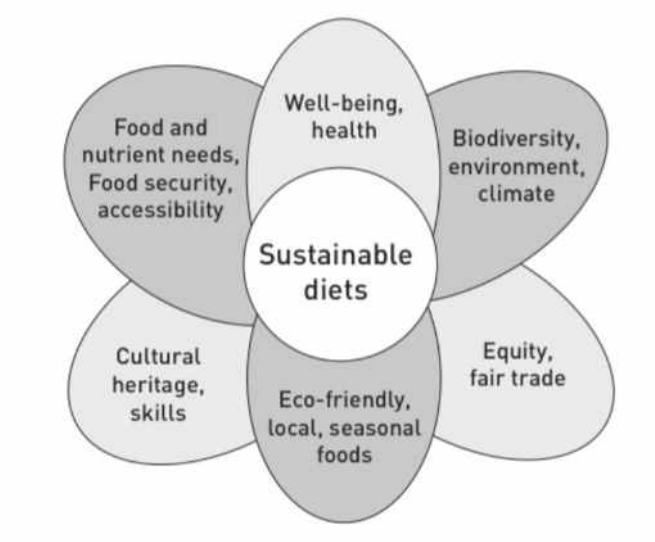


Figura 1.2: Punti cardine del consumo sostenibile

- un sistema di ri-orientamento degli attuali processi di produzione e consumo, senza perdere in efficienza ma orientati a cambiamenti rispetto ai metodi attuali i quali, comunque, vengono emulati anche dai paesi in via di sviluppo;
- l'instauramento di servizi che siano sostenibili dal punto di vista economico, sociale ed ambientale;
- l'uso di prodotti e servizi che possano rispondere ai bisogni primari, migliorando quindi la qualità della vita ma cercando di minimizzare l'uso tanto delle risorse naturali quanto di materiali tossici nonché dell'emissione di sostanze di rifiuto. Naturalmente tutto questo deve avvenire sostenendo pienamento l'uso di forme di energia rinnovabili;
- la promozione di diete sostenibili, le quali devono minimizzare l'impatto sull'ambiente ed essere atte a garantire una nutrizione sicura per una vita sana, non solo per le generazioni attuali ma anche per le future. In modo particolare, le diete sostenibili devono essere:
  - protettive e rispettose delle biodiversità degli ecosistemi
  - accessibili;

- economicamente giuste ed affidabili;
- nutrizionalmente adeguate, sicure e salutari;

Tutto questo, naturalmente, cercando di ottimizzare le risorse umane e naturali.

Tenendo conto di tutto quanto illustrato fin'ora, il mio progetto di tesi è consistito nella realizzazione di un'app per *smartphone* che permettesse al consumatore di pesce di avere un *feedback* sull'ecosostenibilità riguardo al pesce acquistato. Tale ecosostenibilità è basata essezialmente su due punti cardini: **il primo** consiste nel verificare che il pesce acquistato non ricada all'interno della fase di riproduzione per quella specie. Generalmente, ci riferiamo a questo periodo come **Fermo Pesca Biologico** e varia anche in funzione dei mari stessi o zone di essi: non dobbiamo dimenticare infatti che anche il mare ha le sue stagioni che causano l'alternanza di periodi dove è più facile trovare determinati tipi di prodotti ittici e momenti dove è più difficile. Perciò scegliere pesci di stagione, ossia che non sono in fase riproduttiva, offre diversi vantaggi il cui più importante è il consentire alle altre specie di svilupparsi secondo i propri tempi; se poi sono pesci nostrani e provenienti dai nostri mari, si eviterà di farne viaggiare altri, in aereo o su strada, per migliaia di chilometri. A beneficiarne, però, sarà anche il portafoglio, perché il pesce di stagione, in genere, ha un prezzo inferiore.
Il secondo punto cardine riguarda la questione delle taglie minime ossia la lunghezza minima che ogni specie di pesce deve avere affinché possa essere pescato, trasportato e venduto. Infatti, con il Regolamento (CE) n. 1967/2006 del Consiglio, del 21 Dicembre 2006, relativo allo sfruttamento sostenibile delle risorse della pesca, si stabilisce che dal 28 Gennaio 2007 i pesci, crostacei e molluschi inferiori alla taglia minima non possano essere venduti, tenuti a bordo, trasportati, trasferiti o immaganizzati, o anche solo esposti. Secondo i requisiti richiesti per questo progetto, le specie di pesce considerate appartengono maggiormente al Mar Mediterraneo e sono soggette alle restrizioni di pesca stabilite dalla seguente tabella:

Nome comune del pesce	Taglia minima (cm)	Periodo di riproduzione
Sardina	11	da Maggio a Settembre
Acciuga	9	da Aprile a Novembre
Sogliola	20	da Gennaio ad Aprile
Merluzzo o Nasello	20	da Dicembre a Giugno
Gambero Rosa Mediterraneo	2 (lunghezza del solo carapace)	da Aprile a Dicembre
Sgombro	18	Marzo e Aprile
Seppia	15 (raccomandata)	da Aprile a Luglio
Calamaro	12 (raccomandata)	Marzo e Aprile
Triglia di fango	11	da Aprile ad Agosto
Costardella	Non richiesta	da Ottobre a Dicembre

Tabella 1.2: Tabella delle taglie minime e periodi di fermo pesca

Come spiegato nella sezione precedente, il mio progetto di tesi si colloca dentro ad un progetto tanto più grande quanto innovativo che prende il nome di **Citizen Science**, la cui medologia è usata in linea diretta dalla mia collaboratrice, nonché la persona che mi ha commissionato quest'app, **Flavia Bartoccioni** della Facoltà di Scienze di Tor Vergata (si veda la sezione Ringraziamenti per ulteriori dettagli). Questo metodo di ricerca, che si può applicare in diverse discipline, implica il coinvolgimento dei cittadini nella raccolta di dati scientifici. Dunque, il progetto della Dott.ssa Bartoccioni è un progetto di *Citizen Science* poiché, al fine di scoprire la sostenibilità degli acquisti, ci si rivolge ai cittadini i quali si procurano i dati attraverso degli strumenti loro forniti e si preoccupano anche di mandarli secondo le istruzioni ricevute.

Dunque, affinché l'utente possa usufruire di questi dati e possa quindi non solo controllare l'ecodisponibilità del pesce acquistato ma inviare anche delle segnalazioni sui risultati ottenuti, è stata ideata un'app, che d'ora in avanti chiameremo col nome proprio di **Fishmeter**, che potesse permettere all'utente di conseguire tutto questo utilizzandola come descritto dal seguente diagramma dei casi d'uso:

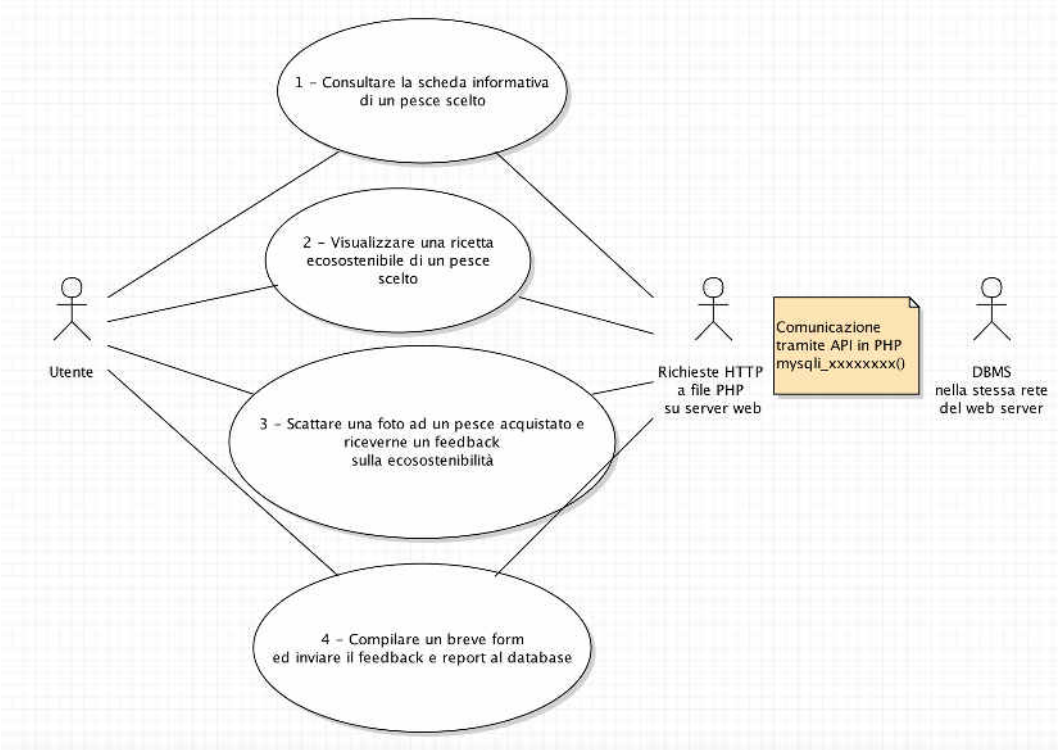
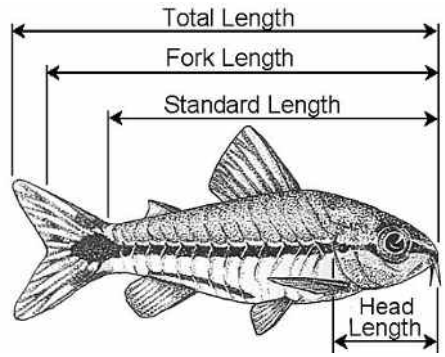


Figura 1.3: Casi d'uso di Fishmeter

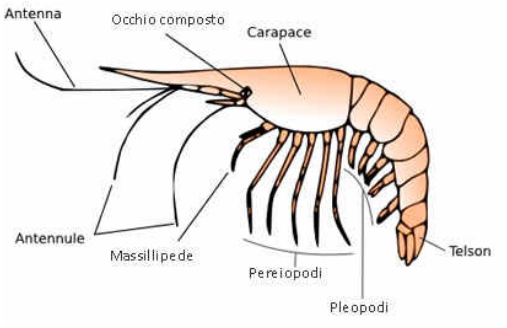
Per meglio comprendere il funzionamento dell'app, in modo particolare della modalità di misura, dobbiamo ricordare che le leggi attuali stabiliscono la modalità standard di misura della taglia del pesce suddividendoli in due grandi famiglie (si veda in proposito la Figura 1.4):

- dei crostacei si misura la sola lunghezza del carapace (il guscio vero e proprio), quindi dagli occhi fino a dove inizia la coda mobile;

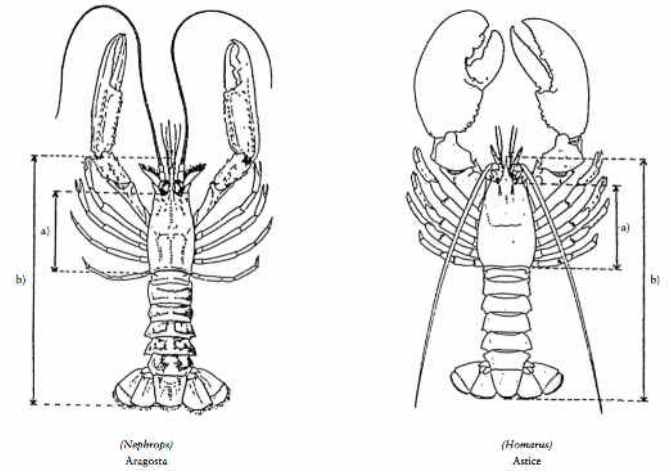
- per tutti gli altri pesci invece si considera la lunghezza dall'estremità anteriore della testa fino all'estremità "maggiore" della pinna caudale.



(a) Taglia del pesce



(b) Taglia del gambero rosa



(c) Taglia di altri crostacei

Figura 1.4: Modalità standard di misura dei pesci

1.3 iOS: Il sistema operativo Apple per tablet ed iPhone

iOS è un sistema operativo sviluppato da Apple atto ad essere eseguito su dispositivi mobili quali iPhone, tablet iPad e iPod Touch. Attualmente, si trova alla versione di rilascio 9.2.1. La versione 9 di questo sistema operativo ha introdotto numerose novità molte delle quali si riflettono come veri e propri *constraints* dal punto di vista dello sviluppatore iOS il quale si deve avvalere per lo scopo dello strumento/software apposito fornito da Apple che porta il nome di **XCode**. Seb-

bene inizialmente lo sviluppo di app per iPhone si effettuasse esclusivamente nel linguaggio Objective-C (una variazione del più noto C++), oggi l'implementazione di app iOS è stata resa molto più semplice e veloce grazie all'introduzione, da parte di Apple, del nuovo linguaggio di programmazione detto **Swift**, che oggi si trova alla versione 2.1, il quale, pur mantenendo sempre un approccio ad oggetti, ha una sintassi molto più semplice (per molti versi è simile al Python) e meno macchinosa dell'Objective-C. Quest'ultimo, infatti, ha una sintassi molto poco familiare e di lungo apprendimento: per esempio non esiste alcuna *dot notation* tipica invece di ogni linguaggio ad oggetti. La mia scelta di sviluppare quest'app in ambiente iOS quindi ricade anche nella mia curiosità ed ambizione personale di imparare ad utilizzare e pandroneggiare al meglio non solo Swift ma anche i nuovi strumenti grafici messi a disposizione dalle ultime versione di XCode per creare tutto ciò che riguarda la GUI (**Graphic User Interface**) di un'app. Infatti, grazie a Swift non è più necessario imparare dettagliatamente il funzionamento di Cocoa (*framework* Apple che si interfaccia tra il tocco delle delle dita dell'utente e l'hardware e che è usato anche per lo sviluppo di interfacce grafiche) a basso livello ma il tutto è molto più accedibile e più velocemente utilizzabile attraverso le nuovi classi e *wrapper* in Swift che, grazie alla sua sintassi semplificata, permette una costruzione più immediata e con meno problemi dei *view controller* e del loro contenuto. In questo senso, di fondamentale importanza è l'implementazione non di un *garbage collector*, che ridurrebbe le *performance* soprattutto in ambiente grafico, ma piuttosto di un meccanismo di ARC (*Automatic Reference Counting*) che si preoccupi direttamente di rilasciare tutte le risorse che rileva come non più utilizzate (in modo particolare quelle grafiche) liberando così anche l'utente dall'oneroso, e spesso problematico, compito di fare la deallocazione di oggetti la cui presenza non è più richiesta e quindi risolvendo la difficile *unsafe pointer management* ed i problemi di *memory leaks* di Objective-C. Altra cosa molto importante in Swift è che la stesura e l'implementazione dei file delle classi è molto simile al Java quando invece Objective-C pretendeva la scrittura di almeno due file per ogni classe: uno con l'intestazione della classe e dei metodi e la lista degli attributi ed un altro con l'implementazione vera e propria dei metodi il che rendeva la gestione di eventuali

modifiche ai file molto più onerosa e soggetta ad errori e dimenticanze. Con Swift invece tutta la nostra classe è contenuta in un unico `file.swift` analogalmente a Java con l'aggiunta che in Swift un file può contenere anche più classi `public` il che è molto di aiuto nella programmazione dell'interfaccia grafica quando si devono trattare assieme molti tipi di classi differenti. Inoltre, in questo senso, il compilatore Swift è in grado di trovare automaticamente molte dipendenze di classi da altri file/classi Swift contribuendo così a ridurre il carico di lavoro al programmatore che invece, con un linguaggio di livello basso come Objective-C, sarebbe stato troppo distratto dallo sviluppo del quadro d'insieme dell'app a causa questi dettagli a basso livello. Tutto questo quindi ci rimanda ai tre scopi fondamentali per cui Apple ha deciso di implementare un nuovo strumento di programmazione:

- *legacy independence*: poiché Objective-C era completamente basato su C/C++, esso non poteva evolvere se prima non evolvevano anche questi ultimi; Swift invece evolve completamente in autonomia in modo che Apple possa concentrarsi di più sul fornire sempre una maggiore velocità di esecuzione;

- modernità e velocità di implementazione: la nuova sintassi di Swift permette di scrivere molto meno codice rispetto all'equivalente in Objective-C;

- sicurezza: sebbene potesse apparire come un vantaggio, in Objective-C non accadeva assolutamente nulla qualora si accedesse ad una variabile, o si richiamare un metodo di un oggetto, settati a `nil`. Questo che era spesso fonte di numerodi *bug* difficili da trovare che portavano ad un malfunzionamento grave dell'app. Swift introduce il tipo di dato `Optional` ossia variabili ed oggetti che possono anche essere `nil` ma qualora vi accedessimo l'applicazione andrebbe in *crash* dando così la possibilità ad XCode di indicarci dove sia avvenuto il problema. Quindi, spetta sempre al programmatore dichiarare nel codice tutte le volte che si aspetta che una variabile o un oggetto sia diverso da `nil` il che, sebbene sembri un inutile lavoro aggiuntivo, porta in realtà ad avere forti garanzie sul corretto funzionamento dell'app una volta che il *bug* è stato risolto.

Ultimo, ma di fondamentale e delicata importanza, va ricordato che con l'avvento di iOS 9, sono stati inseriti numerosi *constraints* nella programmazione delle nuove app, soprattutto con l'arrivo di un nuovo modulo software **ATS** (*App Transport Security*) che richiede certe metodologie di sicurezza qualora l'app che si sta sviluppando necessitasse della comunicazione in rete. Questi nuovi requisiti, assieme ad altri essenziali per il buon funzionamento della mia app, sono riassunti nell'immagine seguente:

Figura 1.5: Requisiti di sistema per l'app Fishmeter

Dunque, ciò che ci interessa sensibilmente del nuovo sistema operativo Apple per smartphone riguarda soprattutto la comunicazione di rete che deve necessariamente essere sicura ed avvenire su una certa versione di SSL; infatti, qualora non rispettassimo i requisiti precedentemente elencati, la nostra app non potrebbe essere pubblicata sull'App Store.

Capitolo 2

Sviluppo dell'app Fishmeter

In questo capitolo illustreremo tutte le fasi attraversate dallo sviluppo di Fishmeter illustrando, per ognuna di esse:

- quale fosse l'idea base che si voleva implementare con
- le relative difficoltà sorte,
- quali problemi di usbailità essa risolvesse e
- quali altri restassero in sospeso fino all'ideazione di una nuova fase successiva.

Tuttavia, prima di questo occorre elencare alcune caratteristiche e scelte implementative comuni ad ogni fase. Nel corso dell'implementazione ho cercanto di mantenermi il piu' fedele possibile al modello MVC (*Model View Controller*). Infatti, possiamo raggruppare tutte le classi Swift di *Fishmeter* nel modo seguente (Figura 2.1):

classi entità : come la classe `Fish` che contiene tutte le caratteristiche del pesce selezionato essenziali al funzionamento base dell'app; oppure la classe `Reporting` che raccoglie invece i dati ottenuti dall'app dopo che quest'ultima ha mostrato il *feedback* all'utente relativo alla sua segnalazione;

classi *model* : dette anche *service*, come la classe `FishService`, si occupano di implementare i metodi di comunicazione in rete e di interrogazione al

database e di ri-arrangiare i dati ottenuti in modo da renderli di facile uso all'interfaccia grafica;

classi *view controller* : ne sono state implementate tante quante le schermate possibili con cui si può interagire con l'app. Il loro compito principale è catturare un evento provocato dall'utente (come il *tapping* su un bottone) ed invocare anche i metodi delle classi *service* precedenti per caricare o spedire dati per poi per ri-passarli alle classi che implementano gli elementi dell'interfaccia grafica vera e propria (sebbene molte di queste "classi" vengano in realtà codificate implicitamente dall'*Interface Builder* di XCode). Inoltre, tali classi prendono provvedimenti per informare l'utente qualora le comunicazioni di rete fornite dalle classi *service* non vadano a buon fine.

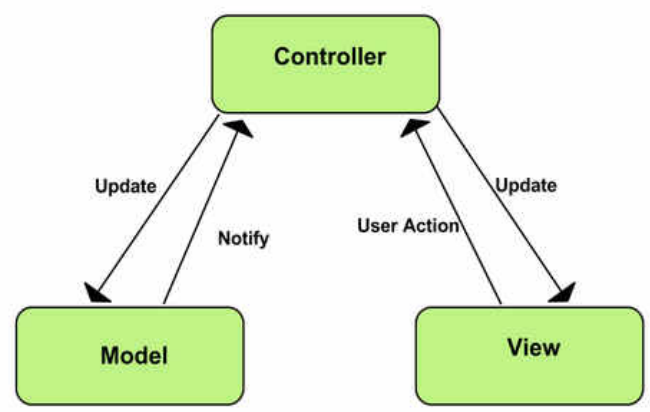


Figura 2.1: `Model-View-Controller` *design pattern*

N.B.: alla fine di ogni fase mostreremo alcuni grafici ottenuti dal *testing*, di quella versione dell'app, su alcuni tipi di utenti diversi. In modo particolare, i pesci su cui si è deciso di testare l'app sono stati l'acciuga e la sogliola, che al momento dell'acquisto in pescheria misuravano rispettivamente circa 113 mm e 265 mm. Si rimanda al capitolo 3 per ulteriori dettagli e spiegazioni in merito ai risutati ottenuti ed al tipo di utenti testati.

2.1 Caratteristiche di base dell'applicazione

Prima di tutto occorre spiegare con quali modalità venga effettuata la stima della lunghezza del pesce. Essenzialmente, per ottenere questo obiettivo sono state considerate due strade possibili: una, più complessa, completamente automatizzata e l'altra, più semplice, che richiede invece più "lavoro" da parte dell'utente. Data la mia inesperienza totale su materie come analisi dell'immagini, o più in generale sulla *computer vision*, si è deciso di seguire la seconda strada per arrivare il prima possibile ad un prodotto completo e funzionante.

La scelta implementativa di un'app non completamente automatizzata, richiede che sia l'utente a dichiarare, in primis, non solo che tipo di pesce sta considerando ma anche, dopo avere scattato la foto, ad indicare sulla stessa il punto di inizio e di fine del pesce stesso. Allo stesso tempo, occorre anche riprendere nella foto anche un'oggetto di riferimento che possa aiutare nello stimare la lunghezza del pesce attraverso un semplice calcolo di confronto/proporzione. La scelta di quest'oggetto è ricaduta sulle monete in quanto sono gli oggetti forse più comuni e che ogni possibile utente sicuramente possiede e che quindi può reperire ed utilizzare facilmente. In questo modo, dopo che l'utente ha dichiarato anche quale moneta è stata utilizzata nella foto e dopo avere indicato la dimensione della stessa, viene allora effettuata una semplice proporzione nel modo seguente:

$$\mathbf{EstimatedRealLength_X} : \mathbf{PhotoLength_{X'}} = \mathbf{RealDiameter_{coin}} : \mathbf{PhotoDiameter_{coin'}}$$

dove:

$EstimatedRealLength_X$: è la stima della lunghezza reale del pesce X che si sta considerando, ciò che vogliamo calcolare;

$PhotoLength_{X'}$: è lunghezza in *Points* del pesce X' nella foto scattata. Ricordiamo fin da adesso che la risoluzione delle immagini nei sistemi iOS non viene considerata in maniera standard con i *Pixels* ma attraverso i *Points* che altro non sono che delle matrici quadrate di pixel che sono tanto più grandi quanto maggiore è la risoluzione del *display* del dispositivo;

$RealDiameter_{coin}$: è la lunghezza reale della moneta utilizzata;

$PhotoDiameter_{coin'}$: è infine la lunghezza stimata del diametro della moneta fotografata, anch'essa misurata in *Points*.

Per dare una maggiore versabilità all'app si è deciso di tenere in considerazione tre tipi di monete: 1€, 2€ e 0.50€ che sono anche le più grandi come dimensioni e che quindi si prestano più facilmente all'ottenimento di una stima del diametro della moneta nella foto, nel modo più facile e preciso possibile. A questo punto, dopo avere ottenuto il valore di $EstimatedRealLength_X$ del pesce considerato, secondo la metodologia appena descritta, non ci resta altro che confrontarlo con la sua taglia minima e vedere se la data odierna, nella quale si scatta la foto e si invia la segnalazione, è compresa nell'intervallo durante il quale si può pescare e consumare il pesce: in altre parole, per avere un *feedback* completamente positivo, la data odierna deve trovarsi al di fuori del periodo di riproduzione del pesce.
A questo meccanismo di base sono state aggiunte via via varie migliorie per mezzo di strumenti *software* non solo per facilitare il compito all'utente nell'ottenere la lunghezza in *Points* dei due oggetti ma anche soprattutto per ottenere la stima della reale misura il più precisa possibile e che avesse un margine di errore di pochissimi millimetri, in eccesso o in difetto.

Per quanto spiegato nel Capitolo 1, risulta evidente come questa app abbia bisogno di una modesta quantità di dati sul pesce selezionato e ne richieda anche il salvataggio di nuovi ottenuti dopo il *feedback* mostrato all'utente: stiamo quindi parlando della comunicazione dell'app con una base di dati che possa risolvere quest'esigenza, come mostrato anche nella Figura 1.2.
Quest'aspetto è stato fin da subito per me un primo passo essenziale di ricerca poiché, trattandosi di argomenti di complessità avanzata, essi di solito non sono compresi nei corsi standard di programmazione per applicazioni mobili. Una prima e semplice considerazione, poteva essere quella di codificare tutti i dati sui pesci in codice Swift o XML all'iterno della stessa applicazione. Tuttavia questo primo approccio risulta completamente inefficiente e di scarsa utilità: non solo perchè ogni modifca dei dati sui pesci richiede una ricompilazione dell'app ma anche perché non fornisce alcuna soluzione alla racolta dati dei *report* e segnalazioni inviati dagli utenti su cui si basa la *Citizen Science*.

Mentre esistono numerose librerie e *wrapper* sviluppate anche in Swift da utenti e programmatori (e disponibili sul portale *GitHub* dal quale citiamo, come i più usati, il progetto **FDMB** (*Flying Meat Database*) e **SQLiteDB**) che permettono una semplice interrogazione di un *DBMS SQLite* integrato direttamente dentro l'iPhone, nella sua versione attuale 2.1.1 Swift non fornisce alcuna *API* o *wrapper* analoghi per l'interrogazione diretta di database remoti. Dunque, per ottenere questo, il tutto si riduce all'utilizzo di classi apposite che permattano l'invocazione di servizi *RESTful* fornite dalla libreria Swift e la cui esecuzione inizia e finsice quindi attraverso il protocollo HTTP ed i suoi metodi di cui, in modo particolare, vedremo come siano stati utilizzati i metodi `GET` e `POST`. Per montare questa struttura quindi occorre la scrittura di alcuni file in PHP residenti su un *web server Apache* che facciano da intermediari tra l'app e il DBMS la cui scelta è ricaduta subito su una versione online di MySQL fornita dal dipartimento di *Computer Science* dell'Università di Bologna. Sebbene a prima vista questo approccio possa sembrare complesso e macchinoso in realtà fornisce un vantaggio di grande valore: Swift, infatti, supporta nativamente `JSON`, il noto formato appositamente creato per l'interscambio di dati tra applicazioni client-server, permettendo un interazione con esso molto semplice e diretta trattandolo come se fosse un comune `array` di oggetti. Inoltre, poiché PHP permette la restituzione dei dati ottenuti da *query* a database nel formato `JSON`, ecco che questa strategia di comunicazione basata su API *RESTful* diventa estremamente vantaggiosa e facile da usare. A titolo dimostrativo, vediamo un paio di questi metodi che permetteono la comunicazione con server e DBMS: il primo che illustriamo implementa una richiesta `HTTP GET` per scaricare i dati basilari di ogni pesce e caricarli nella *home page* dell'app:

Frammento di codice 2.1: Metodo per scaricare i dati dal DBMS online

```
1 internal func requestAllFishes(completionHandler: (success: NSArray?,
    error: NSError?) -> Void) {
2         let finalURL:NSString = "\(Data.webServerFishURL)?all_fishes"
3     self.requestFishesWithHandler(finalURL as String, completionHandler:
          completionHandler)
```

```
4  }
5
6
7  private func requestFishesWithHandler(url:String, completionHandler:
       (success: NSArray?, error: NSError?) -> Void) {
8
9          let configuration =
               NSURLSessionConfiguration.defaultSessionConfiguration()
10
11         let url: NSURL = NSURL(string: url)!
12         let urlRequest: NSURLRequest = NSURLRequest(URL: url)
13
14         let session = NSURLSession(configuration: configuration)
15
16         let task = session.dataTaskWithRequest(urlRequest,
               completionHandler: { (data: NSData?, response:
               NSURLResponse?, error: NSError?) -> Void in
17
18             if (error != nil) { completionHandler(success: nil, error:
                   error) }
19
20             else {
21                 //print(NSString(data: data!, encoding:
                       NSUTF8StringEncoding))
22                 if let responseJSON: [[String: String]] = (try?
                       NSJSONSerialization.JSONObjectWithData(data!,
                       options: NSJSONReadingOptions())) as? [[String:
                       String]] {
23                     completionHandler(success: responseJSON, error:nil)
24              }
25
26              //else { print("ERRORE FORMATO JSON!")}
27             }
```

```
28          })
29
30          task.resume()
31
32  }
```

Possiamo notare nella parte finale del codice come la stringa `JSON` ricevuta dal server venga direttamente convertita in un dizionario chiave-valore di tipo `[String:String]` che viene passato all'interfaccia grafica attraverso un *completion handler* che è fondamentale in questo tipo di metodi. Infatti, in Swift le chiamate a metodi HTTP avviene in modo totalmente asincrono. Ciò significa che la parte grafica dell'app potrebbe iniziare ad accedere a degli *array* di dati prima che questi abbiano finito di caricarsi con l'interrogazione al database, finendo così per sollevare delle eccezioni qualora questi *array* risultassero essere ancora con valore `nil`. È proprio a questo punto che entra in gioco il *completion handler*: il suo blocco di codice viene eseguito solo quando la chiamata asincrona è terminata e quindi in questo modo possiamo garantire di aggiornare l'interfaccia grafica solo all'interno di tale blocco di codice (che in termini di Swift è detto *closure*, simile alle *lambdas functions* di altri linguaggi di programmazione) gestito dal *completion handler*. Dunque, per esempio, all'avvio del caricamento della *home page* dell'app, che altro non è che una tabella (precisamente un `TableViewController` in termini di Swift) le cui *entry* sono i pesci presi in considerazione, avremo questo metodo per scaricare i dati dal DBMS:

Frammento di codice 2.2: Inizializzazione dati dell'home page

```
1  func downloadFishes() {
2          let fishService:FishService = FishService()
3          fishService.requestAllFishes {
4              (response, error) in
5              if (error != nil) {
6                  if (error?.code == -1004) {
7                      dispatch_async(dispatch_get_main_queue()) {
8                          self.presentViewController(Utils.getAlert("Errore
```

```
                        di connessione!", message: "Impossibile
                        comunicare col server. \n Riprovare piu'
                        tardi."), animated: true, completion: nil)
9               }
10          }
11
12              else if (error?.code == -1009) {
13                  dispatch_async(dispatch_get_main_queue()) {
14                      self.presentViewController(Utils.getAlert("Errore
                            di rete!", message: "Connessione ad internet
                            assente. \n Controllare la propria connessione
                            dati."), animated: true, completion: nil)
15                  }
16              }
17          }
18          else {
19              self.fishCollection = fishService.loadFishes(response!)
                    as! [FishIdentity]
20              dispatch_async(dispatch_get_main_queue()) {
21                  self.tableView.reloadData()
22              }
23          }
24      }
25  }
```

Notiamo che l'aggiornamento dell'interfaccia grafica avviene solamente all'interno della *closure* che inizia alla terza riga del precedente Codice 2.2 e che comprende tutto quello da fare appena la chiamata *RESTful* `requestAllFishes()` è terminata. Possiamo vedere come in caso di esito positivo, nell'ultimo ramo `else`, la tabella possa venire caricata e mostrata coi dati appena ottenuti, richiamando il metodo `tableView.reloadData()` che produce il risultato mostrato nella figura seguente:

Figura 2.2: *Screenshot* dell'*home page* dell'app

Concludiamo questa sezione col frammento di codice seguente (Codice 2.3) che mostra invece come funzioni l'inivio di dati al server e da lì al DBMS: semplicemente, invece che effettuare una chiamata `GET` usiamo invece la *POST* preoccupandoci di inserire e codificare tutti i dati raccolti nel `Reporting` all'interno dell'`URL`. Esse si riferiscono direttamente al file PHP che gestisce quell'entità e sono tutte della forma
`https://ltw1600.web.cs.unibo.it/handleFishes.php`:

Frammento di codice 2.3: Funzione RESTful per il salvataggio del report

```
1 internal func addReportService(completionHandler: (success: String? ,
    error: NSError?, data:NSData?) -> Void) {
2         let request = NSMutableURLRequest(URL: NSURL(string:
              Data.webServerFishReportURL)!)
3
```

```
4         request.HTTPMethod = "POST"
5
6         let postParams = "action=save_report
7             &id_fish=\(Reporting.fishId)
8             &fish_name=\(Reporting.fishName)
9             &detectedSize=\(Reporting.detectedSize)
10            &eligibilitySize=\(Reporting.eligibilitySize ? 1 : 0)
11            &eligibilityDate=\(Reporting.eligibilityDate ? 1 : 0)
12            &origin=\(Reporting.origin)
13            &production=\(Reporting.production)
14            &townReport=\(Reporting.townReport)"
15
16        request.HTTPBody =
              postParams.dataUsingEncoding(NSUTF8StringEncoding)
17
18        let saverReport =
              NSURLSession.sharedSession().dataTaskWithRequest(request) {
19            (data, response, error) in
20            if (error != nil) {
21                completionHandler(success: nil, error: error, data:nil)
22            }
23
24            else {
25                completionHandler(success: response!.description, error:
                      nil, data:data)
26            }
27
28        }
29        saverReport.resume()
30    }
```

In questo caso, a livello di chiamante del metodo ci preoccuperemo solo di verificare che la risposta che ci torna indietro dal server sia un `status code`:

`200 OK` o, in caso sfavorevole, un errore a livello di MySQL. In **Appendice A** è possibile visionare alcuni esempi dei file PHP utilizzati a lato server.

Nelle prossime sezioni ci concentreremo in ciò che riguarda lo sviluppo del *core* dell'app ossia all'implementazione delle funzionalità per rispondere ai casi d'uso 3 e 4 della Figura 1.3.

2.2 Prima fase nello sviluppo dell'app: *silly pins*

Le funzionalità centrali dell'app permettono all'utente di eseguire le seguenti operazioni nell'ordine giusto:

1. scattare una foto al pesce scelto assieme ad una moneta;
2. selezionare la moneta usata tramite il relativo pulsante;
3. posizionare due *pins* appositi all'estremità del pesce;
4. posizionare altri due *pins* appositi all'estremità della moneta;
5. ottenere il *feedback*.

Tutte queste funzionalità risiedono quindi nella schermata principale dell'app che è data da un singolo `UIViewController` la quale è una classe fornita dall'*Interface Builder* di XCode che mette a disposizione un'infrastruttura di base per la gestione delle *views* che compongono nel loro complesso la cosiddetta *view interface*. Essa, dunque, è responsabile del caricamento e della coordinazione delle viste in base anche all'interazione che ha l'utente con gli oggetti contenute nelle stesse o anche in base al cambiamento dei dati sottostanti.
Per rispondere al primo punto dei precedenti, si è inserito un `UIImageView`, al centro del *view controller*, che altro non è che un contenitore che permette la visualizzazione di immagini o l'animazione di una serie di esse. In questo contenitore andrà quindi inserita la foto scattata dall'utente il che avviene attraverso l'uso di un `UIImagePickerController`, il tutto sempre già fornito da XCode. L' `UIImagePickerController` avvia automaticamente un'interfaccia standard iOS

per scattare le foto e ci permette il salvataggio della foto o, eventualmente, la riacquisizione della stessa qualora decidessimo che non vada bene.
Ciò fatto, l'utente non deve fare altro che selezionare la moneta che ha utilizzato per la foto appena scattata il che avviene premendo su uno dei tre appositi pulsanti che raffigurano le tre monete il cui uso è consentito dall'app. Va detto, a questo punto, che in tutta l'app vengono mantenute due variabili globali la cui prima è detta `ChosenFish` e mantiene i dati essenziali del pesce (nome comune, taglia minima e stagione riproduttiva) che si è selezionato all'*home page* assieme anche al diametro della moneta scelta. Dunque, la pressione di uno dei tre pulsanti delle monete altro non fa che aggiornare l'attributo `moneyDiameter` di questa classe che verrà poi utilizzato nel calcolo della proporzione finale.
A questo punto, inizia la gestione dei *pins* da muovere sopra la foto. In questa prima fase si è pensato ad essi come delle semplici *UIImageView* di piccole dimensioni, ognuna con la propria icona e colore a seconda che si trattasse o di un *pin* dedicato alla lunghezza del pesce (rosso) oppure di un *pin* dedicato alla gestione del diametro della moneta (blu) come mostra l'immagine seguente:

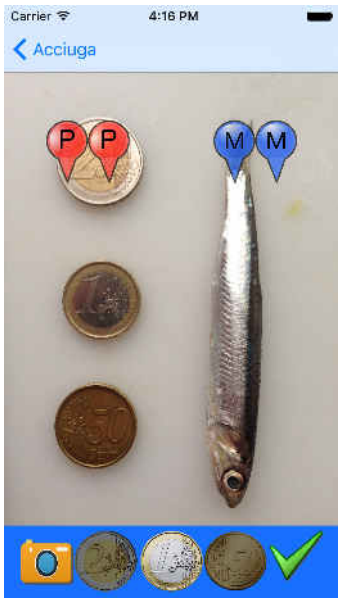


Figura 2.3: *Screenshot* 1 del *core* dell'app (primo step)

Questi *pins* vengono aggiunti come *subviews* al *controller* principale ed ognuno è dotato di un *id:String* così da poterli riconoscere e trattarli singolarmente quando si deve gestire il loro movimento ed i calcoli della proporzione. Per farli muovere all'interno della schermata, si è pensato di fare *override* ed implementare i seguenti metoda:

- `override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?)`

- `override func touchesMoved(touches: Set<UITouch>, withEvent event: UIEvent?)`

- `override func touchesEnded(touches: Set<UITouch>, withEvent event: UIEvent?)`

Questi sono metodi di base per la gestione del tocco da parte dell'utente e sono forniti a tutte quelle classi che estendono la *view* di base, detta *UIView*, e quindi ereditati direttamente anche dalla `UIImageview`. Come suggerisce il loro nome, tali metodi gestiscono i tre stati principali del tocco dell'utente su una vista: l'inizio del *tapping*, l'eventuale l'operazione di *dragging* (trascinamento) della *view* stessa lungo lo schermo, e per finire ciò che bisogna fare appena l'utente rilascia il tocco dallo schermo. In prima battuta, il metodo che ci interessa è `toucheMoved()` il quale aggiorna le coordinate del centro del *pin* che si sta toccando a seconda della nuova posizione del dito dell'utente. In questo modo, quindi, è possibile posizionare i *pins* nella giusta posizione come mostra il seguente *screenshot*:

Figura 2.4: *Screenshot* 2 del *core* dell'app (primo step)

Tuttavia, è subito evidente come questa prima metodologia sia estremamente imprecisa, non solo perchè non fornisce alcun riferimento della posizione di un *pin* rispetto agli altri ma anche perchè l'utente potrebbe posizionarli in modo scorretto: nel caso del pesce se i due *pins* non sono posizionati esattamente uno in linea d'aria dell'altro si otterrebbe una misura maggiore di quella reale in quanto i *pins* si trovano in "diagonale" uno rispetto all'altro fornendo quindi una maggiore distanza tra essi qualora fossero invece allineati. Lo stesso discorso vale esattamente anche per la moneta. Si richiedeva quindi qualche input in più all'occhio che aiutasse l'utente a collocare i *pins* nel modo più corretto possibile senza accorciare o aumentare scorrettamente le distanze tra essi a causa di un posizionamento impreciso. Il metodo che si è pensato per ovviare a questa prima difficoltà è stato quello di tracciare degli assi nel corso di tutta l'operazione di *dragging* dei *pins*. Di questi due assi, uno è parallelo all'asse verticale della foto e passa per il centro del *pin*

mentre il secondo è parallelo all'asse orizzontale della foto è passa per la "punta" in basso dei *pins*. Poichè però noi di ogni *view* conosciamo solo le coordinate del centro (date dall'attributo `self.center`) occorre allora traslare l'asse orizzontale verso il basso esattamente della metà dell'altezza del *pin*. Queste operazioni avvengono attraverso l'uso della libreria `Core Graphics` fornita da Apple che possiamo vedere come una libreria che offre le medisime funzionalità per la *computer graphics* come le note OpenGL e pertanto sono API a basso livello in C/C++ accedibile in modo più semplice dal linguaggio Swift attraverso dei *wrapper*. L'idea di fondo quindi è che, appena l'utente appoggia il dito su un *pin*, compaiano gli assi appena citati e vengano aggiornati nella loro intersezione via via che l'utente si muove sulla *UIImageView* che contiene la foto.

Per fare funzionare tutto questo in modo efficiente, piuttosto che disegnare gli assi direttamente sulla foto è preferibile disegnarli su un'ulteriore `UIView` trasparente e che abbia le stesse dimensioni della foto e che quindi sia posizionata esattamente su essa. In questo modo non solo è possibile lavorare su un'entità separata mantenendo così disgiunte le varie funzionalità, ognuna sulla propria entità, ma possiamo anche gestire in maggiore autonomia la questione degli assi facendoli comparire al bisogno per poi "disporli" e nasconderli qualora essi non siano più richiesti, il tutto senza intaccare o compromettere la *view* della foto.
Affinché la vista che aggiungiamo sia della giusta dimensione, occorre aspettare che iOS abbia applicato tutti i *constraints* grafici e dimensionali al nostro `UIViewController` in modo da avere le dimensioni finali decise per ogni oggetto (secondo i *constraints* indicati nell'*Interface Builder*) qualora ne dovessimo replicare uno. L'applicazione dei *constraints* non avviene subito al caricamento del *controller* quindi non possiamo aggiungere la nuova *view* nel metodo di caricamento di dafault `func viewDidLoad()` ma dobbiamo aspettare che il sistema chiami il metodo `func viewDidAppear(animated: Bool)` all'interno del quale ogni *constraint* è stato applicato e quindi ogni oggetto ha la sua dimensione definitiva. Dunque, facciamo *overriding* di questo metodo all'interno del quale creiamo la *view* dedicata al disegno degli assi che da ora in avanti chiameremo `AxesView`:

Frammento di codice 2.4: Creazione della *view* per il disegno degli assi

```
/* solo qui dentro i constraints sono effettivamente applicati */
override func viewDidAppear(animated: Bool) {

        if (!self.loadedPins) {

            self.loadedPins = true

            self.axisView = AxisView()
            self.axisView.frame = self.imageView.frame
            self.axisView.backgroundColor = UIColor.clearColor()

            self.view.addSubview(axisView)
            axisView.hidden = true

            self.redPinA = PinImageView(imageIcon: UIImage(named:
                "icons/red_pin.png"), location: CGPointMake(40, 110),
                name: "redPinA", axisView: self.axisView)

            self.redPinB = PinImageView(imageIcon: UIImage(named:
                "icons/red_pin.png"), location: CGPointMake(80, 110),
                name: "redPinB", axisView: self.axisView)

            self.bluePinA = PinImageView(imageIcon: UIImage(named:
                "icons/blue_pin.png"), location: CGPointMake(200, 110),
                name: "bluePinA", axisView: self.axisView)

            self.bluePinB = PinImageView(imageIcon: UIImage(named:
                "icons/blue_pin.png"), location: CGPointMake(240, 110),
                name: "bluePinB", axisView: self.axisView)

            view.addSubview(self.redPinA)
            view.addSubview(self.redPinB)
            view.addSubview(self.bluePinA)
```

```
26              view.addSubview(self.bluePinB)
27          }
28  }
```

di cui la linea che ci interessa maggiormente è la n. 9 dove alla nuova *UIView* viene assegnato lo stesso identico *frame* del *UIImageView* principale. Infatti, ogni oggetto presente nell'app in iOS viene identificato con il concetto di `frame` ossia i rettangolo che lo contiene di cui è dato:

- le coordinate dell'origine (ossia lo spigolo in alto a sinistra)
- il valore della larghezza
- il valore della lunghezza

il tutto ovviamente in *Points*. Dunque, nella riga n. 9 facciamo in modo che la *UIView* appena creata non solo abbia la stessa dimensione della *view* che contiene l'immagine ma vi sia anche esattamente sovrapposta. A questo punto quindi occorre che:

- gli assi vengano mostrati in secondo la posizione iniziale del *pin* alla chiamata di `touchesBegan()`
- gli assi vengano aggiornanti man mano che l'utente muove il *pin* nel metodo `touchesMoved()`
- gli assi vengano nascosti quando l'utente ha finito di fare *dragging* e quindi alla chiamata di `touchesEnded()`.

La classe `AxesView` contiene al suo interno un riferimento al *pin* che l'utente sta attualmente considerando (settato all'interno di `tochesBegan`) e man mano che quest'utimo viene mosso viene anche di volta in volta ridisegnata la *view* degli assi a seconda della posizione del *pin* in un determinato istante. L'idea di fondo consiste nel creare un *graphics context* del quale viene fatto un'operazione di *push* all'interno di uno *stack* apposito per passare poi alla *pipeline di rendering*. Affinché questo sia posisbile, occorre che la chiamata a `UIGraphicsGetCurrentContext()`

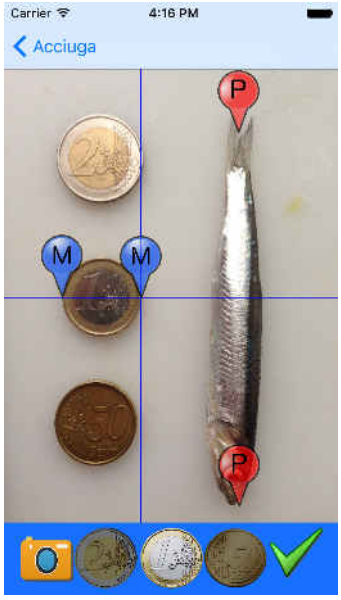
venga effettuata all'interno di un metodo apposito che predisponde la *view* ad essere ri-disegnata: bisogna quindi fare *overriding* del metodo `func drawRect(rect: CGRect)` e mettere al suo interno tutte le chiamate alle API grafiche. Una volta fatto questo si procede a disegnare delle semplici linee da un punto di partenza fino ad un punto di arrivo (la larghezza o altezza assegnate alla *view*) passanti per il centro del *pin* che si sta considerando. Infine, gli assi appena disegnati subiscono il vero e proprio *rendering* mentre il contesto grafico viene rilasciato attraverso la chiamata `CGContextStrokePath()`. Notiamo quindi come queste API siano analoghe a quelle fornite dal framework `GLUT` per le OpenGL, come possiamo vedere dalla seguente classe `AxesView`:

Frammento di codice 2.5: Implementazione della *view* per il disegno degli assi

```
1  class AxisView: UIView {
2      var selected_pin:PinImageView?
3      func drawAxis(fromH:CGPoint, toH:CGPoint, fromV:CGPoint,
           toV:CGPoint) {
4          let context = UIGraphicsGetCurrentContext()
5          switch (selected_pin!.id) {
6          case "redPinA","redPinB":
7              /* set line color as red */
8              CGContextSetRGBStrokeColor(context, 1.0, 0.0, 0.0, 1.0)
9              break
10         case "bluePinA", "bluePinB":
11             /* set line color as blue */
12             CGContextSetRGBStrokeColor(context, 0.0, 0.0, 1.0, 1.0)
13             break
14         default: break
15         }
16         /* line width of axes */
17         CGContextSetLineWidth(context, 1.0)
18         /* draw horizontal axes inside view */
19         CGContextMoveToPoint(context, fromH.x, fromH.y)
20         CGContextAddLineToPoint(context, toH.x, toH.y)
```

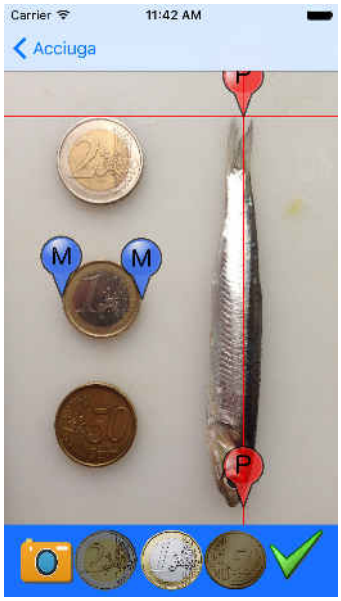
```
21          /* draw vertical axes inside view */
22          CGContextMoveToPoint(context, fromV.x, fromV.y)
23          CGContextAddLineToPoint(context, toV.x, toV.y)
24
25          /* render axes */
26          CGContextStrokePath(context)
27      }
28      override func drawRect(rect: CGRect) {
29          if selected_pin != nil {
30              let fromV:CGPoint =
                    CGPointMake((self.selected_pin?.center.x)!, 0.0)
31              let toV:CGPoint = CGPointMake(self.selected_pin!.center.x,
                    self.frame.height)
32              let fromH:CGPoint = CGPointMake(0.0,
                    self.selected_pin!.center.y - 35.0)
33              let toH:CGPoint = CGPointMake(self.frame.width,
                    self.selected_pin!.center.y - 35.0)
34              self.drawAxis(fromH, toH: toH, fromV: fromV, toV: toV)
35          }
36      }
37  }
```

il che produce il risultato mostrato dai tre *screenshots* seguenti:



(a) Assi di riferimento per la moneta

(b) Assi di riferimento per il pesce



(c) Attenzione a considerare tutta la coda!

Figura 2.5: *Screenshot* 3 del *core* dell'app (primo step)

Mentre il comportamento dei *pin* è regolato dai 3 seguenti metodi:

Frammento di codice 2.6: Implentazione della classe che rappresenta i *pins*

```
class PinImageView: UIImageView {

    var id:String!
    var viewForAxis:AxisView!

    init(imageIcon: UIImage?, location:CGPoint, name:String,
        axisView:AxisView) { ... }

    override func touchesBegan(touches: Set<UITouch>, withEvent event:
        UIEvent?) {
        if let _ = touches.first {
            self.viewForAxis.hidden = false
            self.viewForAxis.selected_pin = self
            self.viewForAxis.setNeedsDisplay()
        }
    }

    override func touchesEnded(touches: Set<UITouch>, withEvent event:
        UIEvent?) {
        self.viewForAxis.hidden = true
    }

    override func touchesMoved(touches: Set<UITouch>, withEvent event:
        UIEvent?) {
        if let touch: UITouch = touches.first {
            self.center = touch.locationInView(superview)
            self.viewForAxis.setNeedsDisplay()
        }
    }
}
```

Sebbene questo metodo provveda ad incrementare la precisione, i risultati sono tuttavia poco soddisfacenti in quanto la misura continua ad essere molto imprecisa,

soprattutto nel contesto della moneta. Infatti, la misura del diametro della moneta sulla foto è troppo grossolano in quanto non si ha alcuna garanzia che l'utente posizioni i due *pins* esattamente alle estremità opposte della moneta per avere un diametro effettivo. Al contrario, è molto più probabile che su ottenga una comune corda di corconferenza che sappiamo essere tutte minori del diametro. In effetti, secondo le prove effettuate, l'utente medio tende sì a posizionare i due *pins* alle estremità della moneta ma essi non corrispondono ad un vero e proprio diametro poiché, il più delle volte, è un segmento di lunghezza minore, sebbene non troppo, sufficiente per fare alterare la misura stimata del pesce, sia in eccesso che in difetto, con una percentuale:

- tra il -5% ed il +11% per l'acciuga, e
- fino al +19% per la sogliola.

Il grafico seguente mostra questi risultati effettuati sui pesci di prova:

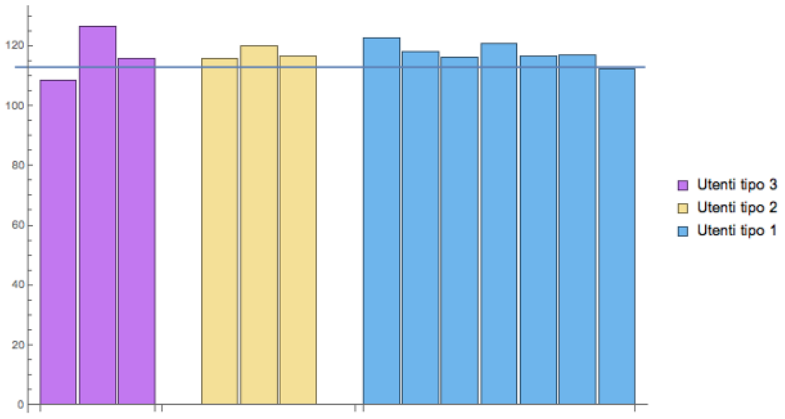


Figura 2.6: Risultati ottenuti con acciuga in fase 1

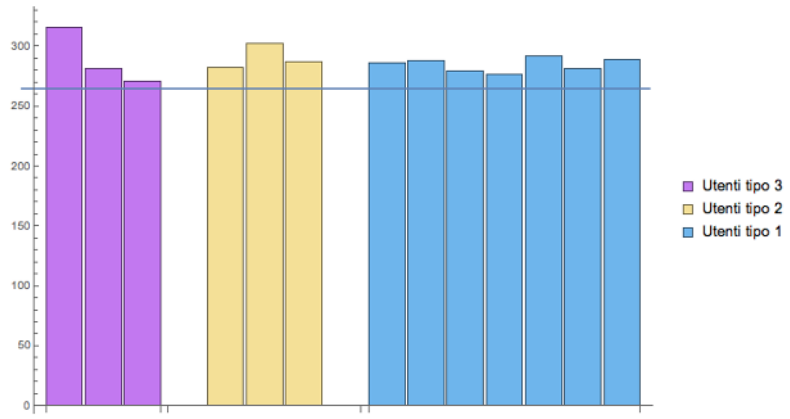


Figura 2.7: Risultati ottenuti con sogliola in fase 1

L'errato posizionamento dei *pins* alle estremità della moneta potrebbe dipendere dalla scarsa visbilità degli oggetti fotografati che questa prima versione dà all'utente e quindi si richiede in primis un miglioramento che fornisca non solo un meccanismo di zoom per posizionare i *pins* esattamente sul confine degli oggetti ma anche un modo per trattare la moneta fotografata esattamente come una vera cironferenza in modo da avere un diametro in *Points* che sia quando più vicino possibile a quello reale.

2.3 Seconda fase nello sviluppo dell'app: *magnifier view*

L'idea di fondo per lo sviluppo di questa seconda versione di `Fishmeter` si ispira ad alcune app per lo *scanning/cropping* di testi ed immagini presenti nell'App Store di Apple. Esse forniscono all'utente una visione della dell'area della foto che stanno considerando più dettagliata e più vicina al suo occhio. Alcune funzionalità di base presenti in questo secondo tenativo di implementazione di `Fishmeter` sono le stesse della fase precedente: il tutto si basa sempre sul posizionamento corretto di alcuni *pins*. Tuttavia, un primo grosso cambiamento consiste nel fornire all'utente un'ulteriore *view* all'interno del quale venga mostrato un'intorno dell'area del *pin* che l'utente sta muovendo, qui e adesso, che sia molto "zoomata" in mo-

do che si possano così distinguere molto chiaramente i vari confini degli oggetti. Chiameremo questa nuova vista fin da subito come `magnifier view` poiché essa si comporta esattamente come una vera e propria lente da ingrandimento, il cui centro è la posizione del *pin* in movimento che si trova quindi sulla stessa retta del virtuale "fuoco della lente". Questo *magnifier* compare non appena l'utente preme su un *pin*, ne viene aggiornato il contenuto al movimento dello stesso e scompare quando l'utente rilascia lo schermo e quindi ripercorre analogalmente i 3 metodi nel Frammento di codice 2.6 della sezione precedente, con la differenza che adesso oltre ad aggiornare la posizione dei centri dei *pins* non dobbiamo più disegnare degli assi ma dobbiamo invece aggiornare e mostrare il contenuto del *magnifier*, una porzione di spazio immagine "zoomato" attorno al *pin* la cui classe viene sensibilmente modificata con due nuovi attributi:

- `viewToBeMagnified:UIView` che, come suggerisce il nome, mantiene un riferimento della vista che bisogna ingrandire e quindi nel nostro caso la `UIImageView` che contiene la foto scattata;
- `magnifier:MagnifyingGlassView` che contiene invece un riferimento alla classe che regola il comportamento della lente *magnifier*.

Prima di discutere del nuovo comportamento dei *pins* dobbiamo capire come funziona la classe del *magnifier* leggendo attentamente il frammento di codice seguente:

Frammento di codice 2.7: Implentazione della classe *magnifier*

```
1 public class MagnifyingGlassView: UIView {
2
3     var viewToMagnify: UIView?
4     var scale: CGFloat = 3.0
5     var zoomedPoint:CGPoint?
6
7     var touchPoint: CGPoint = CGPointZero {
8         didSet {
9             self.center = touchPoint
```

```
10          }
11      }
12
13      required public init(coder aDecoder: NSCoder) {
14          super.init(coder: aDecoder)!
15          commonInit()
16      }
17
18      required public override init(frame: CGRect) {
19          super.init(frame: frame)
20          commonInit()
21      }
22
23      public init() {
24          super.init(frame: CGRectMake(0, 0, size, size))
25          commonInit()
26      }
27
28       private func commonInit() {
29          layer.cornerRadius = frame.size.width / 2
30          self.outlineWidth = 2.0
31          self.outlineColor = UIColor.lightGrayColor()
32          self.size = 200.0
33          layer.masksToBounds = true
34      }
35
36      public override func drawRect(rect: CGRect) {
37          guard let context = UIGraphicsGetCurrentContext()
38              else { return }
39          CGContextTranslateCTM(context, frame.size.width / 2,
                frame.size.height / 2)
40          CGContextScaleCTM(context, scale, scale)
41          CGContextTranslateCTM(context, -zoomedPoint!.x, -zoomedPoint!.y)
```

```
42
43          hidden = true
44          viewToMagnify?.layer.renderInContext(context)
45
46          /* color of axes */
47          CGContextSetRGBStrokeColor(context, 1.0, 0.0, 0.0, 1.0)
48          /* line width of axes */
49          CGContextSetLineWidth(context, 0.5)
50
51          /* draw vertical axis inside magnifier */
52          CGContextMoveToPoint(context, self.zoomedPoint!.x ,
                self.zoomedPoint!.y - (self.frame.size.height / 2))
53          CGContextAddLineToPoint(context, self.zoomedPoint!.x,
                self.zoomedPoint!.y + (self.frame.size.height / 2))
54
55          /* draw horizontal axis inside magnifier */
56          CGContextMoveToPoint(context, self.zoomedPoint!.x -
                (self.frame.size.width / 2), self.zoomedPoint!.y)
57          CGContextAddLineToPoint(context, self.zoomedPoint!.x +
                (self.frame.size.width / 2), self.zoomedPoint!.y)
58          CGContextStrokePath(context)
59          hidden = false
60
61      }
62  }
```

Prima di tutto, notiamo che dentro i costruttori ed in alcuni attributi si eseguono delle operazioni di base di *setup* del *magnifier* come:

- colore e spessore degli assi di riferimento da disegnare all'interno
- valore con cui effettuare l'ingrandimento
- dimensione e forma del *magnifier* stesso.

La posizione del *magnifier* rimane statica per tutto il suo uso, impostata nelle righe 7-11 al punto iniziale di tocco dell'utente sullo schermo rilevato dal metodo `touchesBegan()`. Vedremo come anche questa implementazione sia destinata a fallire poiché continua a proporre misure troppo distanti da quelle reali e per questo si è lasciato il *magnifer* sempre fermo in una posizione predefinita (il centro della schermata) mentre al contrario, in caso di esito positivo delle misure, si sarebbe dovuta aggiornare la posizione del *magnifier* e muoverlo in una porzione di schermo libera, man mano che l'utente muoveva il relativo *pin*, in modo che non gli fosse d'intralcio.
Anche in questo caso "la magia" avviene in buona parte dentro al metodo `drawRect()` chiamato man mano che si deve ri-disegnare il *magnifier* a seguito del movimento del *pin*. Sempre ttraverso le API della libreria *Core Graphics* effettuiamo una prima traslazione all'interno del *magnifier* per considerare una porzione della `UIImageView` sottostante grande tanto quanto le dimensioni della lente. In altre parole, questa prima chiamata ci permette letteralmente di spostare (traslare), di una certa quantità lungo l'asse X e lungo l'asse Y, le coordinate grafiche in spazio utente in modo che il *magnifier* abbia un nuovo sistema di riferimento di coordinate grafiche dato dal suo centro (e quindi effettuando una traslazione di metà larghezza e metà altezza assegnate alla lente). Ciò fatto, eseguiamo un'operazione di scala in positivo di un fattore 3.0 (stabilito a priori e ritenuto sufficiente). Infine, dobbiamo fare una nuova traslazione all'interno del *magnifier* affinché il suo nuovo sistema di coordinate grafiche abbia come centro proprio il centro del *pin*, la posizione in cui quest'ultimo si trova in quell'istante, mentre il resto del *magnifier* sarà un'area data da una certa larghezza ed altezza il cui contenuto è la porzione di immagine attorno al nuovo centro ed ingrandita di tre volte. Inoltre, si è pensato di disegnare anche gli assi cartesiani all'interno della lente così che vi sia continuamente un *mapping* diretto tra il centro della lente stessa ed il centro del *pin* in movimento. La lente con gli assi, in questo modo, può essere di maggiore aiuto all'utente nel posizionare il *pin* esattamente sul confine dell'oggetto ,in modo più preciso possibile. In merito a questo, per dare più visibilità agli assi all'interno della lente, l'icona del *pin* che si sta muovendo viene momentaneamente nascosta

altrimenti apparirebbe ingrandita dentro al *magnifier* e sarebbe di disturbo all'utente. In merito a quest'ultima considerazione, anche l'icona che rappresenta i *pin* è stata cambiata in una più consona ed idonea a questa seconda idea di implementazione di `Fishmeter`, e quindi a forma di croce: verdi per il pesce e blu per la moneta.
Inoltre, poiché la misura della moneta viene fatta in modo completamente diversa rispetto alla prima implementazione, adesso solamente per i *pins* dedicati alla misura del pesce vengono anche disegnati gli assi lungo la foto, sempre per aiutare l'utente a metterli esattamente dritti uno di fronte all'altro.

Tutte le operazioni fin qui descritte vengono effettuate ogni volta che il pin si muove, rilasciando poi il *magnifier* quando l'operazione di *dragging* finisce. Tutto questo è controllato dai nuovi seguenti metodi per il movimento dei *pins*:

Frammento di codice 2.8: Implentazione della classe dei *pins* col *magnifier*

```
override func touchesBegan(touches: Set<UITouch>, withEvent event:
    UIEvent?) {
        if let touch: UITouch = touches.first {

            if ((self.id == "green1") || (self.id == "green2")) {
                self.viewForAxis!.hidden = false
                self.viewForAxis!.selected_pin = self
                self.viewForAxis!.setNeedsDisplay()
            }

            /* hide pin icon --> it won't be showed inside the magnifier
                */
            self.image = nil

            /* set magnifier location and point around which to zoom in
                */
            self.magnifier.touchPoint = self.viewToBeMagnified.center
            self.magnifier.zoomedPoint =
                touch.locationInView(self.viewToBeMagnified)
```

```
16
17              self.viewToBeMagnified.addSubview(self.magnifier)
18              self.magnifier.setNeedsDisplay()
19          }
20  }
21
22
23  override func touchesMoved(touches: Set<UITouch>, withEvent event:
        UIEvent?) {
24          if let touch: UITouch = touches.first {
25
26              if ((self.id == "green1") || (self.id == "green2")) {
27                  self.viewForAxis!.setNeedsDisplay()
28              }
29
30              self.updateMagnifyingGlassAtPoint(touch.locationInView(self.viewToBeMagnifie
31          }
32  }
33
34
35  func updateMagnifyingGlassAtPoint(point: CGPoint) {
36          self.magnifier.zoomedPoint = point
37          self.magnifier.setNeedsDisplay()
38          self.center = point
39  }
40
41
42  override func touchesEnded(touches: Set<UITouch>, withEvent event:
        UIEvent?) {
43
44          if ((self.id == "green1") || (self.id == "green2")) {
45              self.viewForAxis!.hidden = true
46          }
```

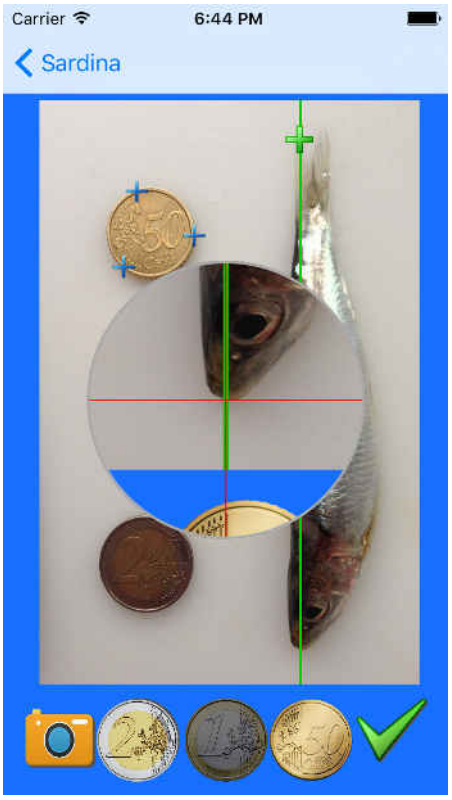
```
47
48          /* restore pin icon when magnifier disapperas */
49          self.magnifier.removeFromSuperview()
50
51          if (self.id.rangeOfString("blue") != nil) {
52              self.image = UIImage(named:"icons/bluePlus.png")
53          }
54          else if (self.id.rangeOfString("green") != nil) {
55              self.image = UIImage(named:"icons/greenPlus.png")
56          }
57
58  }
```

Dunque, ogni volta che l'evento di inizio tocco si verifica, un nuovo *magnifier* viene aggiunto come *subview* alla *view* che deve essere ingrandita, detta (`viewToBeMagnified`).

N.B.: il metodo `drawRect` di una *view* non può mai essere chiamato direttamente dal programmatore ma lo si può invocare attraverso la funzione `setNeedsDisplay()` la quale svolge anche una serie di operazioni a livello di sistema operativo a ridisegna una vista qualora essa sia effettivamente presente e soprattutto visibile nell'attuale schermata (*view controller*) dell'app.

Le tre figure seguenti ci mostrano alcune schermate dell'app con la nuova funzionalità del *magnifier* appena discussa.

(a) Lente per *pin* sulla moneta



(b) Lente ed asse di riferimento per *pin* sul pesce

(c) Posizionamento finale risultante dei *pins* col *magnifier*

Figura 2.8: *Screenshot* 1 del *core* dell'app (secondo step)

La Figura 2.8 (c) ci dà uno spunto interessante per introdurre la nuova modalità con cui viene considerata la misura della moneta in questo secondo *step*. A tal proposito, questa innovazione è basata su una precisa idea geometrica ossia uno dei teoremi fondamentali della circonferenza:

Teorema 2.1. *Per tre punti non allineati passa una ed una sola circonferenza.*

Poiché nella sezione precedente si era argomentato che la disposizione di due soli pin non ci permetteva di avere un diametro esatto in *Points* della circonferenza nella foto, allora si è pensato di aggiungere un terzo *pin* da inserire sul contorno della moneta e così, sfruttando il Teorema 2.1, ottenere un'equazione della circonferenza associata alla moneta da cui poi ottenere il diametro. Dunque, l'utente non deve fare altro che posizionare i tre *pins* sul bordo della moneta, dopodiché l'*app* eseguirà semplici conti di algebra lineare per ottenere l'equazione della circonferenza, la quale sappiamo essere della forma: $x^2+y^2+ax+by+c=0$. Poiché noi conosciamo le coordinate del centro dei tre *pins* non ci resta altro che valutare tre volte l'equazione appena citata sostituendo alle variabili x, y rispettivamente il valore di ascissa e coordinata del centro del *pin*. In questo modo otteniamo un sistema lineare di tre equazioni in tre incognite della forma:

$$\begin{cases} ax_1 + by_1 + c = d_1 \\ ax_2 + by_2 + c = d_2 \\ ax_3 + by_3 + c = d_3 \end{cases}$$

dove $d_i = -(x_i^2 + y_i^2)$ e x_i, y_i si riferiscono alle coordinate del centro del *pin* i-esimo

Dunque non resta altro che trovare un metodo per risolvere questo sistema lineare, motivo per il quale ci viene incontro il *frame* `Accellerate` sviluppato da Apple. Il compito di questa libreria è fornire al programmatore delle *routine* in C non solo per i calcoli matematici (soprattutto algebra lineare) ma anche per *signal and image processing*. Su queste basi, implementiamo la seguente funzione che risolve il sistema:

Frammento di codice 2.9: Risoluzione di un sistema lineare in Swift

```
1 func solve( A:[Double], _ B:[Double] ) -> [Double] {
```

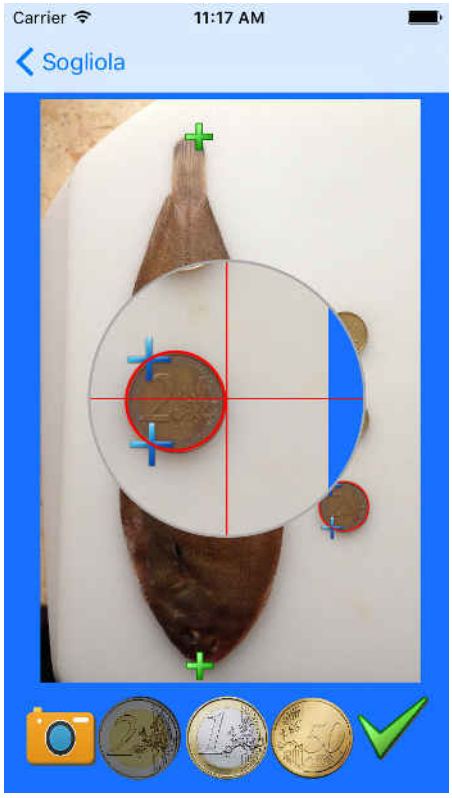
```
2         var inMatrix:[Double]     = A
3         var solution:[Double]     = B
4         // Get the dimensions of the matrix. An NxN matrix has N^2
5         // elements, so sqrt( N^2 ) will return N, the dimension
6         var N:__CLPK_integer      = __CLPK_integer( sqrt( Double( A.count
              ) ) )
7         // Number of columns on the RHS
8         var NRHS:__CLPK_integer   = 1
9         // Leading dimension of A and B
10        var LDA:__CLPK_integer    = N
11        var LDB:__CLPK_integer    = N
12        // Initialize some arrays for the dgetrf_(), and dgetri_()
              functions
13        var pivots:[__CLPK_integer] = [__CLPK_integer](count: Int(N),
              repeatedValue: 0)
14        var error: __CLPK_integer = 0
15        // Perform LU factorization
16        dgetrf_(&N, &N, &inMatrix, &N, &pivots, &error)
17        // Calculate solution from LU factorization
18        _ = "T".withCString {
19            dgetrs_( UnsafeMutablePointer($0), &N, &NRHS, &inMatrix,
                  &LDA, &pivots, &solution, &LDB, &error )
20        }
21        return solution
22    }
```

dove A è la matrice dei coefficienti $\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix}$ mentre B è il vettore colonna dei termini noti $\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$. Dopo averle indicato con `"T"` la modalità di lettura delle matrici per righe, la *routine* `dgetrs_( )` si preoccupa di risolvere il siste-

ma lineare utilizzando la cosiddetta **Decomposizione LU** (la quale fattorizza la matrice in una triangolare superiore, una triangolare inferiore ed una di permutazione) per calcolarsi il determinante e quindi trovare il valore delle tre incognite. Al termine di questo procedimento possiamo calcolarci le coordinate del centro della circonferenza come $c = (-\frac{a}{2}, -\frac{b}{2})$ mentre il diametro sarà dato dalla formula $\sqrt{(\frac{a}{2})^2 + (\frac{b}{2})^2 - c}$ e quindi risolvere la proporzione.

(a) Circonferenza ottenuta attorno alla moneta (sogliola)

(b) Circonferenza ottenuta attorno alla moneta (sardina)



(c) Zoom sulla circonferenza ottenuta in (a)

Figura 2.9: *Screenshot* 2 del *core* dell'app (secondo step)

Sebbene la Figura 2.9 (c) ci mostri che la circonferenza ottenuta è piuttosto precisa rispetto alla moneta fotografata, tuttavia i grafici seguenti sulle misure effettuate dagli utenti su sardina e sogliola si rivelano ancora poco imprecisi e non soddisfacenti. Infatti,

- con l'acciuga si ha una sovrastima fino al +5% circa, mentre
- con la sogliola si ha una sovrastima maggiore del +8%.

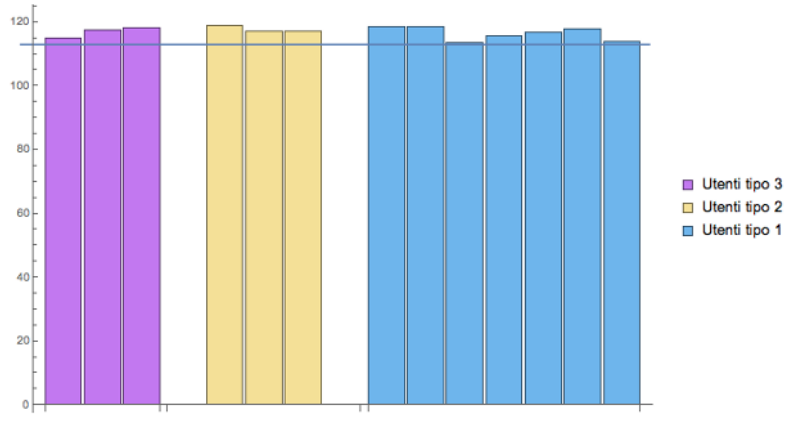


Figura 2.10: Risultati ottenuti con acciuga in fase 2

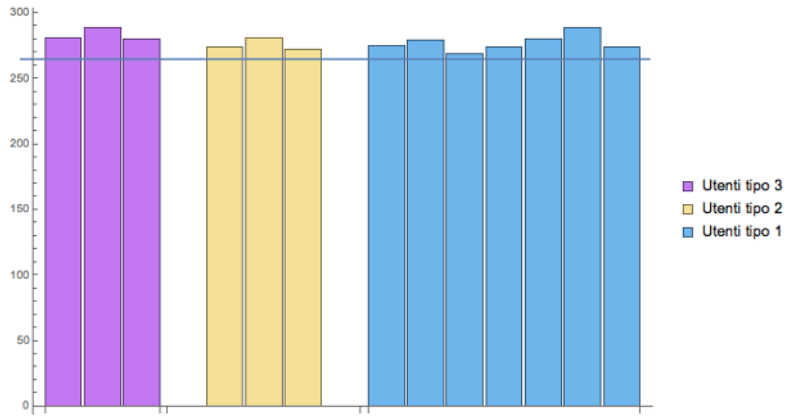


Figura 2.11: Risultati ottenuti con sogliola in fase 2

Notiamo infatti come la misura continui ancora ad essere sovrastimata di circa mezzo centimetro, sebbene si sia comunque ottenuto un leggero miglioramento rispetto alla prima fase.

In seguito ad un'attenta analisi dell'immagine ci si è accorti che la quantità in eccesso deriva ancora principalmente dalla misura della moneta che continua ad essere troppo approssimativa. Infatti, bisogna tenere conto che comunque l'utente cerchi di mantenere il cellulare in posizione orizzontale, **la forma circolare della moneta non verrà mai preservata nella foto**. Al contrario, essa diventerà più simile ad un' ellisse, con la dimensione orizzontale maggiore di quella verticale e che quindi non possiamo approssimare semplicemente ad una circonferenza. Inoltre, è proprio la larghezza dell'ellisse quella che invece dovremmo considerare in quanto è quella che preserva il vero diametro della moneta quando la foto viene fatta tenendo lo smartphone inclinato, anche se di poco. Tutto questo porta alla decisione che occorra fare un terzo sviluppo dell'app tenendo in considerazione i seguenti punti:

- l'utente deve essere "forzato" a mentenere lo smartphone quanto più orizzontale possibile nel momento in cui viene scattata la foto;

- l'utilizzo di *pin* non solo si rivela ancora insoddisfacente, soprattutto nella misura della moneta, ma presenta anche dei problemi di usabilità in quella del pesce. Infatti, ricordando la Figura 2.4, l'operazione di *dragging* di *pins* ha dimostrato essere molto scomoda ed ambigua da posizionare non solo quando bisogna considerare l'estremità posteriore del pesce - in quanto andrebbe posizionato non ad un'estremità della pinna - come sarebbe d'intuito - ma in mezzo alle due ed aiutandosi con gli assi come vediamo in Figura 2.5 (c) - ma anche perchè occorre fare estrema attenzione ad allineare esattamente i due *pins* per evitare di avere una "misura in diagonale" che sarebbe maggiore di quella reale il che continua a provocare una sovrastima nella proporzione. Dunque questa modalità deve essere completamente scartata in favore di un'idea completamente ex novo che garantisca non solo una precisione maggiore ma che sia anche meno "macchinosa" per l'utente medio il quale non è sicuramente disposto a muovere un grande numero di *pins* e per di più facendo una così estrema attenzione a posizionarli sul contorno.

2.4 Terza fase nello sviluppo dell'app: *pincher views*

Per rispondere al primo degli ultimi due punti della sezione precedente, si è deciso di avvalersi di un'interfaccia, fornita da iOS, per accedere ai cosiddetti *motion services* che ci forniscono i dati relativi "movimento" e posizione del *device* rispetto ad un sistema di riferimento. In modo particolare, con quest'interfaccia possiamo accedere a:

- dati dell'accellerometro;
- informazioni sulla *rotation-rate* e sull'orientamento del dispositivo attraverso la lettura del giroscopio;
- dati dal magnetometro;

dove ciò che risponde alle nostre esigenze è elencato nel punto 1 e viene implementato dalla classe `CMDeviceMotion` presente nel framework Apple ***Core Motion***. In modo particolare, ciò che dobbiamo estrarre da questa classe è la misura della gravità lungo l'asse z la quale varia a seconda che l'utente mantenga l'iPhone in modo più o meno inclinato orrizontalmente e tale gravità viene misurata sottraendo l'accellerazione che imprime l'utente al dispositivo dall'accellerazione globale dello stesso. Perciò, quello che occorre fare prima di tutto è corredare l'interfaccia dell'`UIImagePickerController` per scattare la foto al pesce con questo strumento, detto *motion manager*, per fare in modo che, a seguito di un frequentessimo aggiornamento nel tempo dello stesso, un messaggio di *warning* sia inviato all'utente qualora mantenga il dispositivo troppo inclinato nel momento in cui sta scattando la foto. Questo si traduce nell'avviare la seguente funzione su un *thread* separato non appena l'*UIImagePickerController* ha finito di caricarsi:

Frammento di codice 2.10: Programmazione dell'accellerometro

```
1 func startMotionManager() {
2
```

```
3         let alertController = UIAlertController(title: "Attenzione!",
              message: "Mantieni l'iphone il piu' orizzontale possibile!",
              preferredStyle: .Alert)
4
5         //var showAlert:Bool = false
6
7         self.motionManager.deviceMotionUpdateInterval = 0.01
8         self.motionManager.startDeviceMotionUpdatesUsingReferenceFrame(CMAttitudeRefere
              toQueue: NSOperationQueue.currentQueue()!, withHandler:{
9             deviceManager, error in
10            print(deviceManager?.gravity)
11            /* se inclino troppo o troppo poco l'iphone */
12             if !((deviceManager?.gravity.z >= -0.999) &&
                   (deviceManager?.gravity.z <= -0.991)) {
13                if ((!alertController.isViewLoaded()) ||
                      (alertController.view.window == nil)) {
14                    self.imagePicker.presentViewController(alertController,
                          animated: true, completion: nil)
15                }
16            }
17            /* se l'iphone e' inclinato bene allora nascondo l'alert */
18            else {
19                if ((alertController.view.window != nil) ||
                      (alertController.isViewLoaded())) {
20                    alertController.dismissViewControllerAnimated(true,
                          completion: nil)
21                }
22            }
23        })
24    }
```

Avviando l'applicazione direttamente da XCode, mantenendo il dispositivo connesso al computer e facendosi stampare i valori della gravità lungo l'asse z

mentre si inclinava più o meno lo smartphone in posizione orizzontale, ne è risultato che, per cercare di minimizzare l'inclinazione in modo ottimale, essa deve essere compresa nell'intervallo $[-0.999... - 0.991]$, Qualora l'utente inclinasse troppo il telefono uscendo da questo range allora viene mostrato a schermo un semplice *alert* che disabilita momentaneamente le funzioni fotografiche invitando l'utente a regolare il dispositivo nel modo più orizzontale possibile.
N.B.: quest'accorgimento serve anche ad evitare che, nello scattare la fotografia, venga catturato anche lo spessore della moneta il che indurrebbe l'utente in errore a considerarlo come parte della superficie della moneta stessa: avrebbe così una misura del diametro dell'ellisse molto maggiore di quella reale portando, questa volta, ad una sottostima della misura del pesce nel calcolo della proporzione.
Per risolvere il vero problema dell'applicazione si è pensato invece che la soluzione più pratica e meno soggetta ad errore è quella di applicare una semplice operazione di *pinching* di due vere e proprie *views* da posizionare una sulla moneta e l'altra sul pesce.

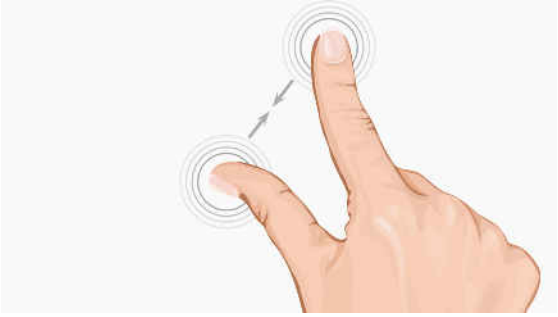


Figura 2.12: Operazione di *pinching* con le dita

Così facendo, si introduce in questo progetto un movimento delle dita (Figura 2.12) che risulta molto comune ed intuitivo in molte app e di cui Apple fornisce numerosi strumenti per riconoscerli e gestirli, liberando così l'utente dalla seccatura di posizionare correttamente i *pins* uno rispetto all'altro sebbene il requsito della precisione
rimane presente e fondamentale. A tal proposito ci viene ancora in aiuto la classe `UIView` la quale definisce un'area rettangolare nello schermo e relative interfacce

per poterne manipolare forma e contenuto. Perciò, non appena la foto è stata scattata, l'app inserisce due *views* - una di forma rettangolare per il pesce ed una di forma ellittica per la moneta - in posizioni centrali allo schermo lasciando all'utente il compito di posizionarle e ridimensionarle come mostra la Figura 2.13. Come accennato, il meccanismo base di dimensionamento di queste *views* è dato dall'inserimento di un `UIPinchGestureRecognizer` il quale rimane in attesa dell'evento dato dall'appoggiare due dita sulla *view* stessa tale che:

- se la vista considerata quella relativa al pesce, esso ne provoca un ridimensionamento solo in verticale;
- se la vista considerata è quella ellittica allora *pinch gesture recognizer* ne gestisce un aggiustamento in entrambe le dimensioni.

Figura 2.13: Le due *view* adibite alla misura

In questo modo, l'utente riesce molto intuitivamente a posizionare le due viste in modo rapido e **quasi** definitivo ed il tutto è regolato semplicemente da una trasformazione affine di scala. Tuttavia, occorre anche fornire all'utente degli ulteriori strumenti che gli permettano di modificare le dimensioni delle viste di quantità più piccole in modo da farle combaciare poi il più possibile con gli estremi dei due oggetti, il tutto supportato e a seguito di operazioni di *zoom in*. Prima di tutto, tali operazioni di *zoom in/out* sono ottenibili gratuitamente inserendo semplicemente tutto il contenuto della schermata dell'app in un *controller* che nasca di default con delle funzionalità di *scrolling* e nel nostro caso si tratta semplicemente di un *UIScrollView*. Dopodiché si è pensato di fornire all'utente due modalità di aggiustamento delle *view*:

1. una, più semplice ed immediata, basata sul *dragging* dei 4 *pins* che stanno ai vertici della vista;

2. un'altra, leggermente più impegnativa ma più precisa, che consenta l'incremento ed il decremento di unità fondamentali (*Points* nel caso di Apple) lungo una certa dimensione della vista alla volta.

2.4.1 Ridimensionamento della vista tramite *dragging* di *pins*

Questa funzionalità permette l'incremento e decremento della vista fluidamente e simultaneamente anche lungo due direzioni diverse, orizzontale e verticale. Per ottenere quanto ci mostra le figura precedente e per quanto detto in sezione 2.2, occorre posizionare i 4 *pins* rispettivamente in:

- nello spigolo in alto a sinistra ossia all'origine del *frame* della vista;

- nello spigolo in alto a destra della vsta dato dalla somma delle coordinate dell'origine con la larghezza;

- nello spigolo in basso a sinistra dato dalla somma delle coordinate dell'origine con l'altezza;

- nello spigolo in basso a destra dato dalla somma delle coordinate dell'origine con l'altezza e larghezza.

A questo punto, tutto il gioco si svolge nelle mani dei *pins* regolati dalla classe `PinImageView` il cui compito è duplice:

1. potersi muovere all'interno della schermata;
2. man mano che si muovono allargare ed allungare (o viceversa) la vista a cui fanno riferimento a seconda della direzione in cui si muovono e rispetto alla loro posizione nella vista, ossia al vertice della stessa cui appartengono.

In questa terza versione dell'app, per il movimento delle viste o dei *pins* si è deciso di usare esclusivamente l'`UIPanGestureRecognizer` in quanto non solo presenta una maggiore sensibilità al tocco dell'utente rispetto agli altri metodi standard ma anche perché offre un maggiore controllo sugli stati del *recognizer* in modo da potere meglio svolgere alcune azioni qualora il *recognizer* entri in funzione. Una di queste, nonché quella più importante, è quella di disabilitare le funzionalità di *scrolling* mentre l'utente sta facendo *dragging* della vista o dei *pins* in modo che queste tre operazioni non vadano in conflitto tra loro, per poi riattivarle quando il *pan gesture recognizer* viene rilasciato.
Il ridimensionamento della vista a seconda del movimento dei *pins* avviene attraverso queste fasi:

1. riconoscimento di quale *pin* l'utente sta utilizzando in base al suo id che banalmente indica il suo spigolo di appartenenza;
2. aggiornare le coordinate del centro del *pin* al nuovo punto di arrivo;
3. ottenere i parametri di scala (in positivo ed in negativo) da applicare alla vista, in modo da
4. costruire un nuovo frame ad assegnarlo alla vista;
5. aggiornare la posizione degli altri *pins* coinvolti per effetto collaterale, riapplicando quanto detto all'inizio di questa sezione.

Di queste fasi quella critica è la numero 2 ed è descritta dalla seguente funzione:

Frammento di codice 2.11: Implementazione del movimento dei *pins* per adattare una vista

```
1  func handlePan(recognizer: UIPanGestureRecognizer) {
2          if (self.panRec.state == UIGestureRecognizerState.Began) {
3              self.scrollContainer.scrollEnabled = false    }
4
5          else if (self.panRec.state == UIGestureRecognizerState.Ended) {
6              self.scrollContainer.scrollEnabled = true
7          }
8
9          let translation = recognizer.locationInView(self.container)
10         let parametersScale:(o:CGPoint, sx:CGFloat, sy:CGFloat) =
               self.getScaleParameters(translation)
11         self.viewToScale.frame = CGRectMake(parametersScale.o.x,
               parametersScale.o.y, self.viewToScale.frame.width +
               parametersScale.sx, self.viewToScale.frame.height +
               parametersScale.sy)
12         self.viewToScale.updatePinsPosition()
13
14 }
15
16 func getScaleParameters(arrivalPoint:CGPoint) -> (o:CGPoint,
       sx:CGFloat, sy:CGFloat) {
17         var scaleX:CGFloat = 0.0
18         var scaleY:CGFloat = 0.0
19
20         switch (self.id) {
21
22         case "up\_left":
23             /* store in origin the actual origin point of the pincher
                   view */
24             let newOrigin:CGPoint = arrivalPoint
```

```
25            let oldOrigin:CGPoint =
                  CGPointMake(self.viewToScale.frame.origin.x,self.viewToScale.frame.origi
26            if (oldOrigin.x < newOrigin.x) {
27                scaleX = -(newOrigin.x - oldOrigin.x)
28            }
29            else {
30                scaleX = oldOrigin.x - newOrigin.x
31            }
32            if (oldOrigin.y < newOrigin.y) {
33                scaleY = -(newOrigin.y - oldOrigin.y)
34            }
35            else {
36                scaleY = oldOrigin.y - newOrigin.y
37            }
38            return (newOrigin,scaleX, scaleY)
39
40        case "up\_right":
41            let oldCorner:CGPoint =
                  CGPointMake(self.viewToScale.frame.origin.x +
                  self.viewToScale.frame.width,
                  self.viewToScale.frame.origin.y)
42            if (oldCorner.x < arrivalPoint.x) {
43                scaleX = arrivalPoint.x - oldCorner.x
44            }
45            else {
46                scaleX = -(oldCorner.x - arrivalPoint.x)
47            }
48            if (oldCorner.y < arrivalPoint.y) {
49                scaleY = -(arrivalPoint.y - oldCorner.y)
50            }
51            else {
52                scaleY = oldCorner.y - arrivalPoint.y
53            }
```

```
54              return (CGPointMake(self.viewToScale.frame.origin.x,
                    arrivalPoint.y), scaleX, scaleY)
55
56          case "down\_left":
57              let oldCorner:CGPoint =
                    CGPointMake(self.viewToScale.frame.origin.x,
                    self.viewToScale.frame.origin.y +
                    self.viewToScale.frame.height)
58              if (oldCorner.x < arrivalPoint.x) {
59                  scaleX = -(arrivalPoint.x - oldCorner.x)
60              }
61              else {
62                  scaleX = oldCorner.x - arrivalPoint.x
63              }
64              if (oldCorner.y < arrivalPoint.y) {
65                  scaleY = arrivalPoint.y - oldCorner.y
66              }
67              else {
68                  scaleY = -(oldCorner.y - arrivalPoint.y)
69              }
70              return (CGPointMake(arrivalPoint.x,
                    self.viewToScale.frame.origin.y), scaleX, scaleY)
71
72          case "down\_right":
73              let oldCorner:CGPoint =
                    CGPointMake(self.viewToScale.frame.origin.x +
                    self.viewToScale.frame.width,
                    self.viewToScale.frame.origin.y +
                    self.viewToScale.frame.height)
74              if (oldCorner.x < arrivalPoint.x) {
75                  scaleX = arrivalPoint.x - oldCorner.x
76              }
77              else {
```

```
78                scaleX = -(oldCorner.x - arrivalPoint.x)
79            }
80            if (oldCorner.y < arrivalPoint.y) {
81                scaleY = arrivalPoint.y - oldCorner.y
82            }
83            else {
84                scaleY = -(oldCorner.y - arrivalPoint.y)
85            }
86            return (self.viewToScale.frame.origin, scaleX, scaleY)
87        }
88 }
```

A titolo puramente esemplificativo, discuteremo il comportamento solo del pin in alto a sinistra nella vista che nel codice viene chiamato `up_left`. Muovendo questo specifico *pin*, l'origine del *frame* della vista viene ad essere modificato nonché la sua altezza e/o larghezza a seconda della direzione in cui esso si sposta, cosa che possiamo dedurre a seconda delle nuove coordinate assegnate al centro del *pin* stesso:

- se l' ascissa è aumentata (il *pin* si sposta verso destra), allora occorre decrementare la larghezza della vista;
- se l' ascissa è diminuita (il *pin* si sposta verso sinistra), allora occorre incrementare la larghezza della vista;
- se l' ordinata è aumentata (il *pin* si sposta verso il basso), allora occorre decrementare l'altezza della vista;
- se l' ordinata è diminuita (il *pin* si sposta verso l'alto), allora occorre incrementare l'altezza della vista.

Quindi, i valori con cui aggiustare la vista vengono calcolati per mezzo di addizioni o sottrazioni tra le diverse ascisse o le ordinate ed eventualmente, come nel caso di questo specifico *pin*, viene calcolata anche una nuova origine oppure viene restituita quella attuale. La figura seguente mostra il funzionamento del

codice di quanto appena spiegato immaginando di muovere il *pin* cerchiato verso lo spigolo in alto a sinistra dello schermo:

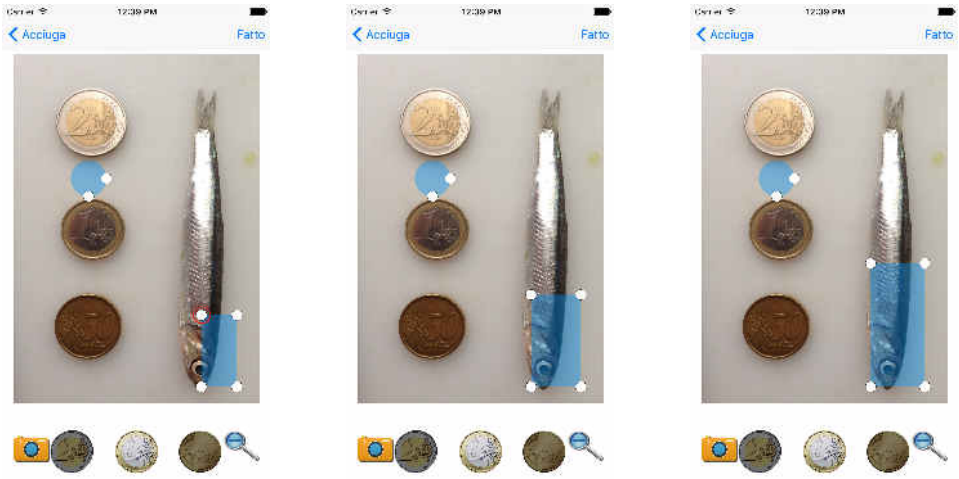


Figura 2.14

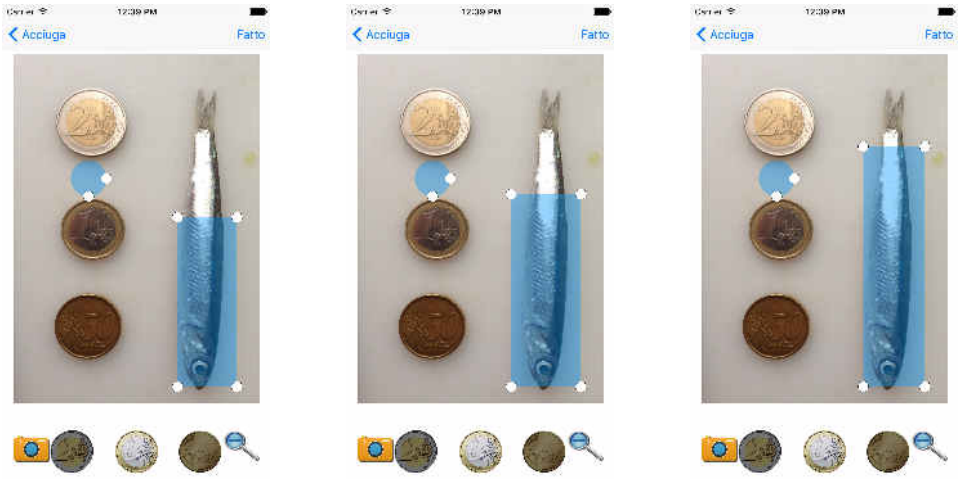


Figura 2.15: Funzionamento di un *pin* di aggiustamento della vista

2.4.2 Ridimensionamento della vista tramite strumenti di maggiore precisione

Ovviamente, la stessa identica procedura, appena discussa nella sezione precedente, può essere applicata anche alla *view* ellissoidale da posizionarsi sulla mone-

ta. Tuttavia, per aggiustare quest'ultima si è pensato ad un meccanismo ancora più sofisticato e più sensibile che permettesse di ricoprire la vista sulla moneta in modo quasi perfetto, tenendo conto del ruolo fondamentale e delicato che gioca il diametro della moneta fotografata in quest'app. Questa nuova idea si distacca completamente dal ridimensionare una vista tramite *dragging* di *pins* ma permette invece l'incremento o il decremento di una piccola quantità alla volta, 0.5 *Point*, lungo una sola dimensione e direzione ed attraverso la ripetuta pressione su un apposito pulsante. Questa funzionalità è stata implementata in un'apposita *view* indipendente, detta `FittersContainer`, e viene attivata e mostrata attraverso un `UILongPressGestureRecognizer`. Quest'ultimo è un riconoscitore di eventi che mostra la vista `FittersContainer` quando si fa un *tapping* prolungato di due secondi sulla vista ellittica, come mostrano i seguenti *screenshots*:

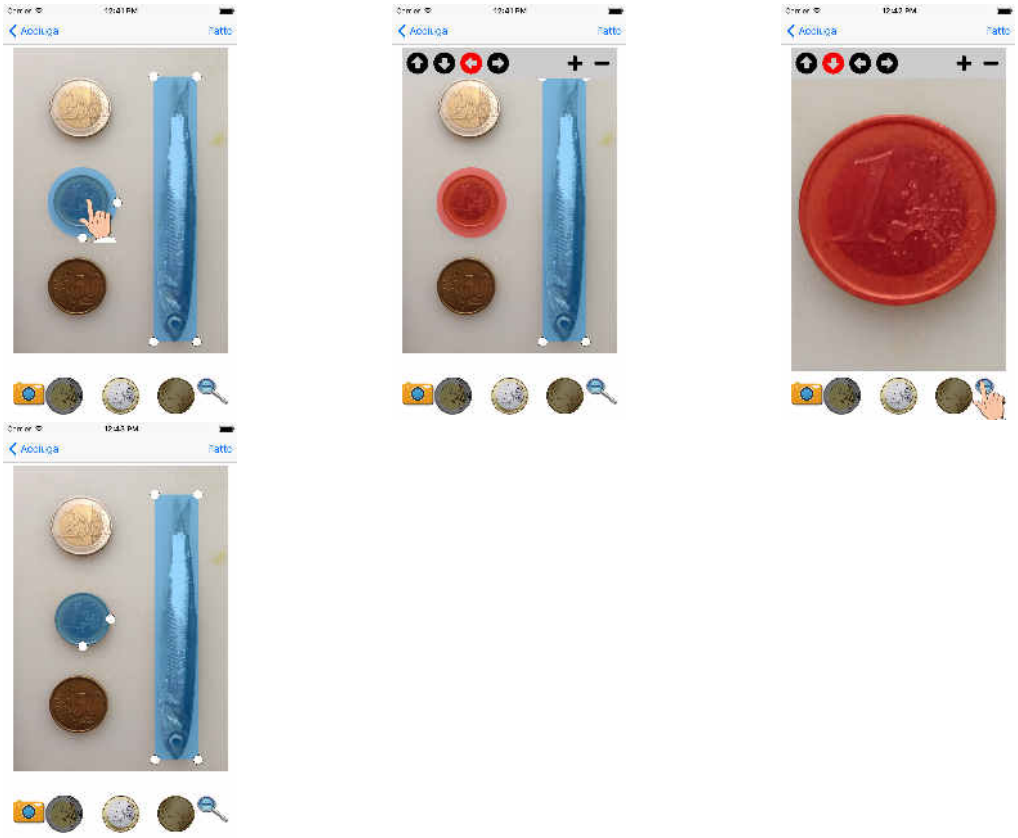


Figura 2.16: *Screenshot* del funzionamento della vista di aggiustamento

N.B.: la terza schermata in Figura 2.16 ci fa pensare al perché del pulsante di *zoom out* con l'icona della lente. Infatti, senza quest'utimo strumento, una volta che l'utente ha finito di ridimensionare una vista sulla moneta in seguito ad un'operazione di zoom, se vuole passare ad esaminare il pesce l'unico modo per tornare alla schermata principale sarebbe quello di fare un *pinch out* sulla foto, movimento opposto a quello fatto per lo *zoom in*. Questa *gesture*, se la moneta è a tutto schermo come nell'esempio, potrebbe essere catturata invece dalla *view* sulla stessa la quale verrebbe accidentalmente rimpicciolita e quindi l'utente dovrebbe fare tutto di nuovo da capo. Per evitare questo, si è pensato di mettere un pulsante apposito in modo da ritornare alla scena iniziale senza compromettere alcuna *pincher view* appena impostata.
Poichè questo funzionamento è stato esteso anche alla vista rettangolare usata per il pesce, si è pensato di colorare di rosso quella vista che è attualmente attiva sotto al controllo della `FittersContainer`. Ovviamente, sebbene l'utente lo possa utilizzare come meglio crede, questo strumento è stato pensato per essere richiamato in seguito ad un' ampia operazione di *zoom in* in modo che la moneta riempia lo schermo dell'iPhone il più possibile (Figura 2.16 di destra): solo così infatti si può adattare la vista alla moneta il più possibile. Il suo funzionamento è semplice ed auto-esplicativo: una volta attivata, basta selezionare prima la direzione lungo cui si vuole ridimensionare la vista in rosso e poi incrementarla o decrementarla di 0.5 *Points* attraverso i pulsanti "più" e "meno". Una volta che l'utente è soddisfatto, può o passare a ridimensionare l'altra vista (sempre premendovi sopra per due secondi) oppure disattivare completamente la funzionalità premendo per due secondi in qualsiasi altro punto della foto. L'implementazione di questa funzionalità è molto semplice ed è molto simile al Frammento di codice 2.11: semplicemente, la classe che la regola contiene il riferimento alla vista attualmente in rosso e a quale pulsante delle quattro direzioni è stato attivato, riconoscibile sempre attraverso degli id. Inoltre, i pulsanti "più" e "meno" sono controllati da una funzione detta `adjustViewPincher(sender: UIButton)` che fa semplicemente un unico grande *switch* considerando l'id del pulsante direzione attivato mentre, come prima, si eseguono sempre operazioni di addizione e sottrazione sull'altezza o la larghezza

della vista rossa ma sempre di una quantità prefissata di 0.5 *Points*. Infatti, se si osserva attentamente la Figura 2.16, si noterà che non è possibile applicare sulla vista in rosso il *dragging* dei *pins* per ridimensionarla, i quali però vanno pur sempre aggiornati nel momento in cui si nasconde la `FittersContainer` e si ritorna alla modalità standard. Nel prossimo capitolo, il finale, mosteremo i risultati ottenuti con questa nuova modalità e trarremo le conclusioni.

2.4.3 Sviluppo delle parti finali dell'app

Prima di concludere, un'ultima parola sull'implementazione è meritata riguardo allo sviluppo delle funzionalità finali. Nell'introduzione e nel primo capitolo abbiamo detto che l'utente, dopo avere scattato la foto, deve ricevere l'esito del suo acquisto di pesce sottoforma di due *feedback*:

- se il pesce è lungo almeno quanto la sua taglia minima;
- se il pesce è stato acquistato nel giusto periodo, ossia non in quell'intervallo di mesi in cui si trova nella stagione riproduttiva.

Sulla base di questo, una volta che l'app ha eseguito il calcolo della proporzione, si controlla se il mese della data attuale in cui viene scattata la foto rientra nell'intervallo di riproduzione di quella specie. In caso di risultato sia negativo, l'app si preoccupa di interrogare nuovamente il database e di fornire un piccolo elenco all'utente di quali, invece, siano le specie di pesce che egli può consumare in quel mese, purché siano sempre maggiori della taglia minima prevista (si veda la Figura 2.17).

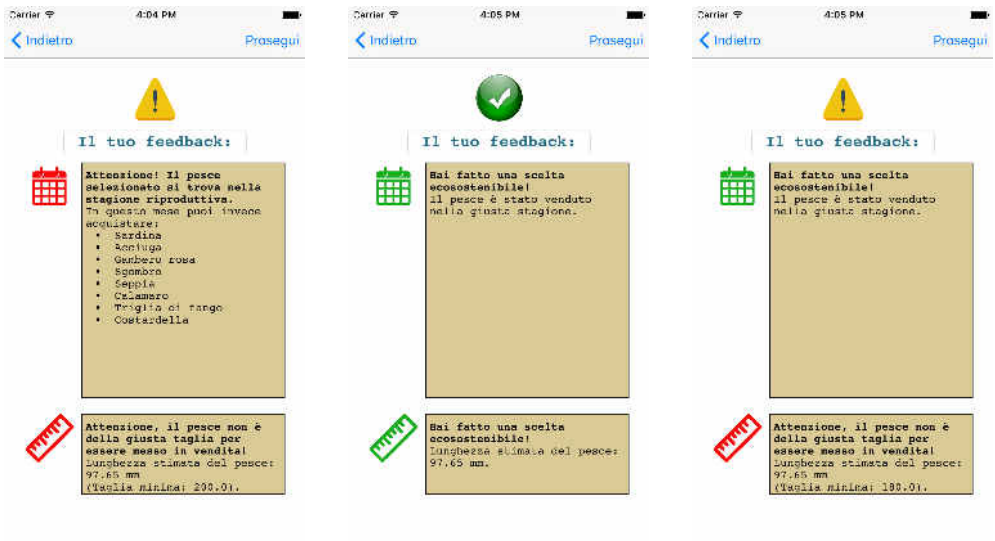


Figura 2.17: *Feedback* dell'esito della segnalazione mostrato all'utente

Infine, attraverso il pulsante `Prosegui`, l'utente è invitato ad inserire le informazioni finali per completare il proprio *report*. Esse sono:

- provenienza del pesce acquistato (Mar Mediterraneo o Altro);
- modalità di produzione con cui il pesce acquistato è stato trattato, da scegliere tra:
  - allevamento;
  - pescato fresco;
  - sconosciuto;
- comune di residenza del cittadino che invia la segnalazione.

In iOS, poiché non esistono né *radio buttons* né *check buttons*, per permettere all'utente di inserire dei dati pre-selezionati si usano i cosiddetti `UIPickerView` che possiamo vedere come una versione dei classici menù a tendina ma con una forma e funzionamento simile al "tamburo della pistola", anch'essi molto usati nelle app più comuni grazie alla loro facilità d'uso molto intuitiva e versatile. Dunque, sono stati inseriti tre *picker data* per permettere all'utente di inserire i dati richiesti

elencati precedentemente. In modo particolare, si vuole mettere in evidenza il *data picker* relativo al punto tre. A differenza degli altri due, questo *picker* presenta due sezioni: nella prima sezione sono contenuti i comuni di interesse per l'inivio dei *reports* mentre nella seconda sezione abbiamo le relative frazioni. Il comportamento interessante di questo *picker* è che è possibile cambiare il contenuto della seconda componente al variare del contenuto della prima. In questo modo, piuttosto che visualizzare un unico listato di tutti i luoghi tutti insieme e senza criterio, appena l'utente seleziona il comune di appartenenza gli vengono mostrati subito anche le sue frazioni nella parte destra del *data picker*. Tutto questo è regolato da due metodi da implementare dell'interfaccia `UIPickerView`: il primo stabilisce cosa il *picker* debba mostrare nella seconda sezione al variare del contenuto della prima mentre il secondo metodo stabilisce come esso debba comportarsi quando viene selezionato il contenuto della prima sezione e cosa mostrare nella varie etichette (Figura 2.18) dopo la selezione dell'utente. A tal proposito si veda il seguente frammento di codice:

Frammento di codice 2.12: Implementazione del *data picker* per il comune di residenza

```
1  static let comuneResidenzaData:[String] = ["- Seleziona comune - ",
       "Monte Argentario", "Orbetello", "Porto Santo Stefano", "Porto
       D'Ercole", "Albinia", "Ansedonia", "Fonte Blanda", "Giannella",
       "Quattro Strade", "San Donato", "Talamone"]
2
3  func pickerView(pickerView: UIPickerView, didSelectRow row: Int,
       inComponent component: Int) {
4    if (self.hasSection) {
5      if (component == 0) {
6        if (row == 0) {self.selectedComponent = "- Seleziona comune -"}
7        if (row == 1) {self.selectedComponent = "Monte Argentario"}
8        if (row == 2) {self.selectedComponent = "Orbetello"}
9        self.myPicker.reloadComponent(1)
10      }
11
```

```
12      if (component == 1) {
13        if (self.selectedComponent == "Monte Argentario") {
14          self.selectedData.text = "\(self.pickerData[row + 2]) (Monte
                Argentario)"
15        }
16        else {
17           if (self.selectedComponent == "Orbetello") {
18             self.selectedData.text = "\(self.pickerData[row + 4])
                   (Orbetello)"
19           }
20        }
21       }
22    }
23          ...
24 }
25
26
27 func pickerView(pickerView: UIPickerView, viewForRow row: Int,
       forComponent component: Int, reusingView view: UIView?) -> UIView {
28
29        var pickerLabel = view as? UILabel;
30
31        ...
32
33        if (self.hasSection) {
34            if (component == 0) {
35                pickerLabel?.text = self.pickerData[row]
36            }
37            else {
38                if (self.selectedComponent == "Monte Argentario") {
39                    pickerLabel?.text = self.pickerData[row + 3]}
40                else {
41                    if (self.selectedComponent == "Orbetello") {
```

```
42                      pickerLabel?.text = self.pickerData[row + 5]
43                  }
44                  else {pickerLabel?.text = ""}
45              }
46          }
47      }
48
49      ...
50
51      return pickerLabel!;
52  }
```

Perciò nel primo metodo, una volta che è stata selezionata la prima componente (detta in realtà componente zero) richiamiamo il metodo `self.myPicker.reloadComponent(1)` che ricarica la seconda componente (detta in realtà la uno) di destra col contenuto appropriato. Tale metodo in realtà è un *wrapper* per richiamare il secondo metodo nel Frammento 2.12 che provvede a mostrare le opzioni giuste nella parte destra del *data picker* ossia le frazioni relative del comune scelto. La figura seguente mostra un'esecuzione di questo codice:

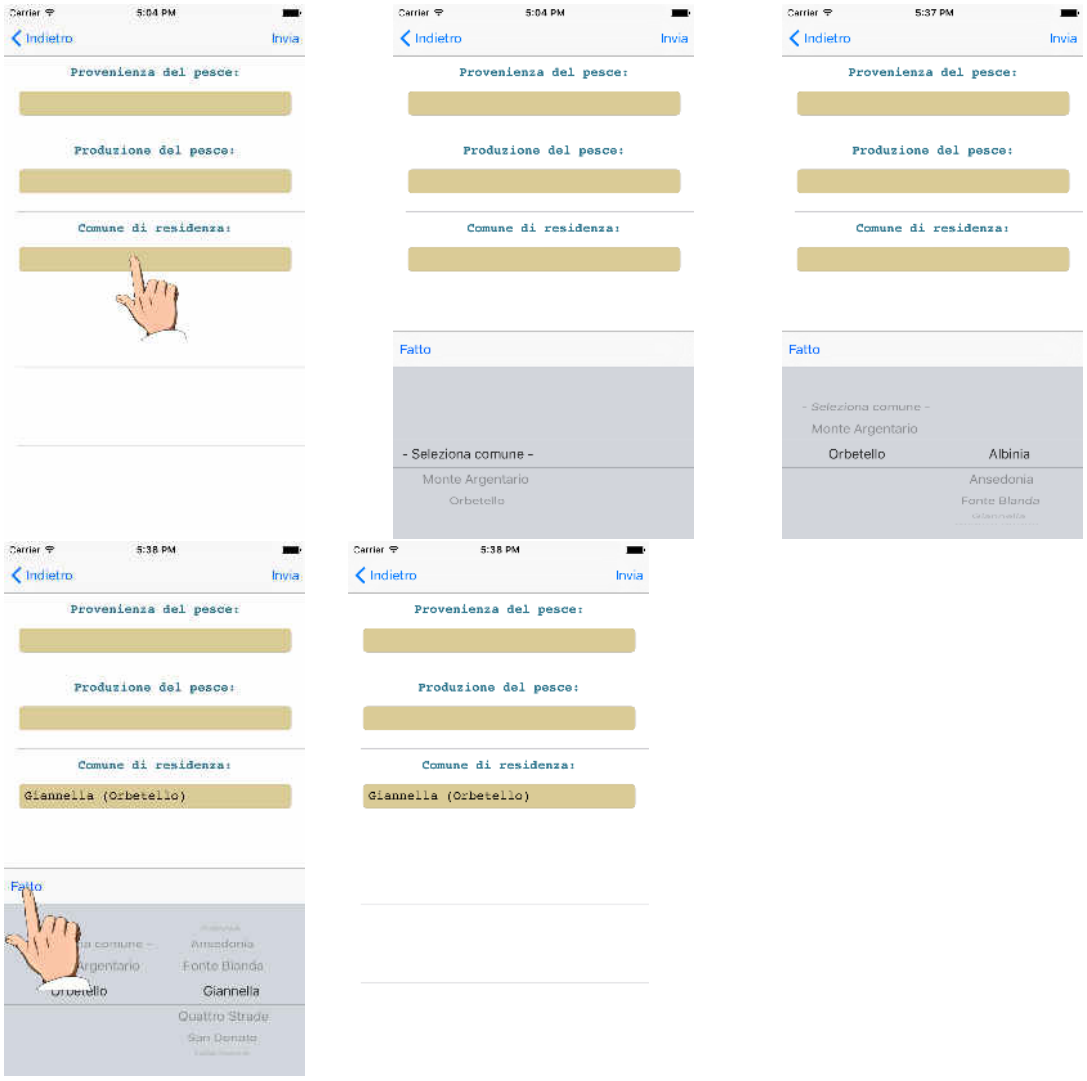


Figura 2.18: Funzionamento del *data picker* per il comune di residenza nell'ultima schermata dell'app

Infine, premendo sul pulsante `Invia` tutti i dati raccolti vengono inviati al database con una richiesta `HTTP POST` eseguendo il codice indicato in Frammento 2.3.

N.B.: gli errori e le eccezioni gestiti in tutta l'app dall'app riguardano essenzialmente problemi nella comunicazione di rete e sono i seguenti:

- connessione di rete assente nel dispositivo;

- errore di comunicazione con il server web;
- controllo del corretto formato JSON ricevuto in seguito ad una *query* `SELECT` al DBMS;
- errore verificatosi nell'esecuzione di una *query* MySQL.

Capitolo 3

Conclusioni

3.1 Analisi dei risultati

Dalle Figure 2.6-2.7 e 2.10-2.11 e da quelle seguenti in questa sezione possiamo vedere come vi sia stato un graduale aumento della qualità del servizio fornita dall'app. Perciò, i risultati ottenuti alla fine della terza fase di sviluppo sono stati ritenuti abbastanza soddisfacenti da potere considerare questa fase come la conclusiva e quella definitiva per quest'app.

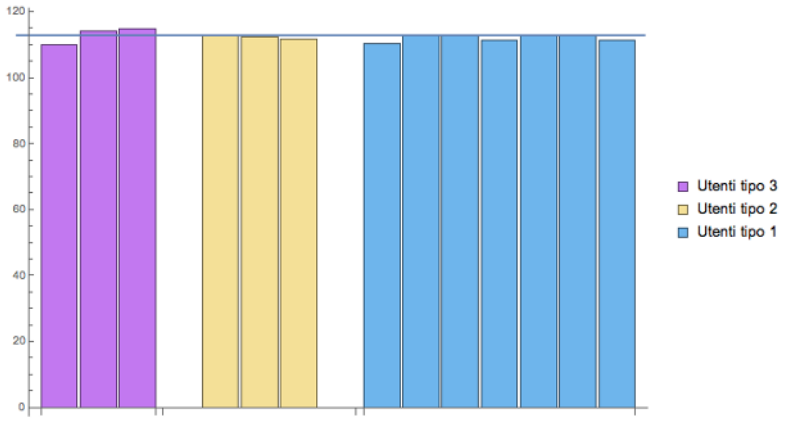


Figura 3.1: Risultati ottenuti rispetto all'acciuga in fase 3

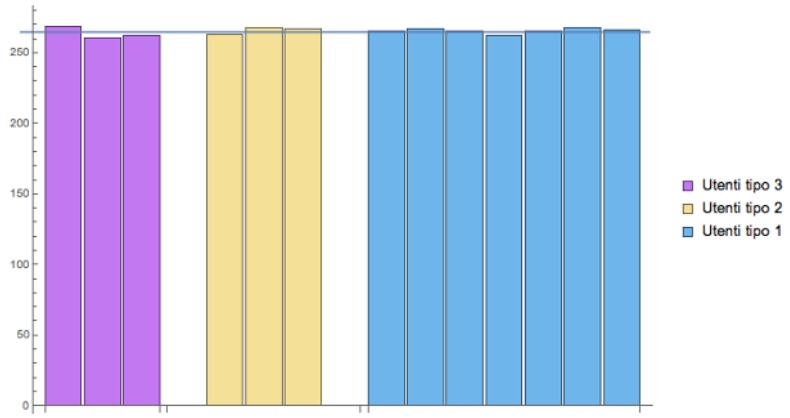


Figura 3.2: Risultati ottenuti rispetto alla sogliola in fase 3

Dunque, come mostrano le precedenti Figure 3.1 e 3.2, grazie all'utilizzo delle *pinching views* la stima della taglia del pesce è sovra(o sotto)stimata solo di piccole quantità ritenute trascurabili, almeno per un tipo di utente. In effetti, si è ritenuto di testare l'app, in tutto il suo corso di sviluppo, sottoponendola all'uso di tre diverse tipologie di utenti:

tipo 3 : sono persone solitamente poco predisposte all'uso di smartphone e quindi di app;

tipo 2 : sebbene non informatici, sono tuttavia persone abituate a lavorare con strumenti di precisione (in pratica trattasi di geometri).

tipo 1 : sono persone che usano smartphone ed app senza difficoltà, e comprendono sia informatici sia no.

In base a questi risultati e ad un'attenta osservazione del comportamento degli utenti durante l'attività di test, si può affermare che quest'app si rivolge ad un tipo di *target* utenti ben delineato e non è quindi per tutti. In un primo momento si potrebbe limitare il *target* utenti solamente al tipo 1 sopra elencato, ma non è esattamente così. Infatti, si è rilevato anche come quest'app non sia immediatamente *user friendly* e quindi non di uso auto-esplicativo ma richieda invece in primis una panoramica generale sul suo funzionamento ed in seguito una certa attenzione e predisposizione all'utente non solo ad utilizzarla con calma e con precisione ma

soprattutto a prendersi una certa quantità di tempo relativamente lunga per portare a termine tutti i *tasks* del'app. Perciò, il tempo di esecuzione dell'app non rientra nei requisiti principali di usabilità e ne viene quindi sconsigliato l'uso ad utenti che sono maldisposti ad una certa concentrazione o che hanno troppa fretta. Infatti, non di rado si è notato una certa difficoltà tra gli utenti (anche quelli di tipo 1) a ridimensionare le *views* sopra agli oggetti muovendo i *pins* mentre sembrano dimenticare facilmente non solo la possibilità di fare *zoom in* della foto ma anche quella di utilizzo degli strumenti di precisione illustrati nella sezione 2.4.2. Tutto questo quindi porta a molteplici ri-esecuzioni delle funzionalità del *core* dell'app che permettono di avere un *feedback* sulla misura. Si potrebbe richiedere quindi l'implementazione di alcuni messaggi di *pop-up* che spieghino il funzionamento di base dell'app al primo avvio; tuttavia, poiché l'app verrà presentata all'interno di un progetto più ampio con tanto di documentazione online, si rimanda questa funzionalità all'implementazione di un'adeguata sezione del relativo sito Internet che spieghi bene all'utente le modalità d'uso. Nonostante queste difficoltà, tuttavia la stima della misura della taglia del pesce è stata ritenuta molto soddisfacente tanto in ambito informatico quanto in quello (bio)ecologico senza bisogno di dovere tenere in considerazione necessariamente una "fascia grigia" dei pesci, ossia quelli che si trovano leggermente sotto-taglia e quindi si dà per buono il *feedback* sulla misura ricevuto dall'app sebbene spesso sia leggermente approssimativo.

3.2 Sviluppo futuro: riconoscimento automatico con OpenCV

Tenendo conto delle limitazioni e delle difficoltà nell'uso dell'app spiegate nella sezione precedente, un'importante sviluppo futuro di questo lavoro sarebbe quello di svincolare l'utente il più possibile da posizionare e ridimensionare le viste sugli oggetti fotografati. Questo richiede quindi il riconoscimento automatico, da parte dell'app, sia del pesce che della moneta nella foto che l'utente ha scattato. In questo campo, entra in gioco **OpenCV**, il noto *framework opensource* creato per la *computer vision* il cui scopo, ricordiamo, "è creare un modello approssimato del

mondo reale (3D) partendo da immagini bidimensionali (2D). La visione artificiale deve riprodurre la vista umana. Vedere è inteso non solo come l'acquisizione di una fotografia bidimensionale di un'area ma soprattutto come l'interpretazione del contenuto di quell'area. L'informazione è intesa in questo caso come qualcosa che implica una decisione automatica" {cit. Wikipedia}

Questo *framework* è attualmente alla versione 3.1 ed esiste per tutti i sistemi operativi, compresi quelli per smartphone con l'unica differenza che, essendo scritto in C++, non è possibile utilizzarlo in Swift con i nuovi strumenti di sviluppo Apple ma bisogna ricorrere a scrivere dei *wrapper* nel vecchio Objective-C i quali contengono del codice in C++ vero e proprio. Tra le varie componenti di questo *framework* ve ne è una molto importante che cerca di riconoscere oggetti di interesse in foto o anche in *stream* video. Questo processo è diviso in due fasi: *training* e *classification*. Nella prima fase si usa una particolare *subroutine* di OpenCV che, dato un insieme di foto dell'oggetto da riconoscere, dette ***positives***, ed un insieme di immagini di *background* che non contengano l'oggetto di interesse, dette ***negatives***, a seguito di una lunga fase di allenamento, produce in *output* un particolare file detto `classifier.xml`. Quest'ultimo file può essere passato ad alcune API OpenCV le quali cercheranno di riconoscere l'oggetto di interesse nelle foto che andranno ad esaminare, e questo processo è appunto detto *classification*.

L'esperimento che ho tentato di portare a termine in quest'app era almeno il riconoscimento della moneta nella foto scattata dall'utente ed il posizionamento corretto della vista, della dimensione più o meno giusta, sopra di essa. Per fare questo mi sono avvalso di numerosi tutotrial in rete e del forum ufficiale di supporto ad OpenCV ed indicati nella bibliografia.
Prima di tutto mi occorrevano numerose immagini dell'oggetto da riconoscere e quindi ho deciso di iniziare col girare un piccolo video (di circa 10-15 secondi) ad una moneta da 2€ utilizzando lo smartphone e cercando di mantenerlo il più orizzontale e vicino alla moneta. Dopodiché, avvalendomi di una serie di API OpenCV per l'analisi dei video, ho estratto ogni fotogramma del video e ne ho ritagliato solo la moneta ottenendo circa 200 immagini come la seguente:

Figura 3.3: Moneta da 2€ ricavata da *strem* video

Dopodiché, era necessario procurarsi un grosso archivio di immagini da usare come negativi le quali non dovevano contenere alcuna moneta. Tutti i *tutorial* seguiti segnalano a questo proposito il seguente link online:
`http://tutorial-haartraining.googlecode.com/svn/trunk/data/negatives/`.
Da questo archivio è possibile scaricare ben oltre 3000 immagini generali da utilizzare come sfondo all'oggetto da riconoscere le quali sono principalmente foto di paesaggi, cantieri o testi scritti come la seguente:

Figura 3.4: Esempio di negativo utilizzato nell'algoritmo di riconoscimento di OpenCV

A questo punto occorre creare una grossa quantità (molte migliaia) di vere e proprie immagini di allenamento: con questo tipo di immagine intendiamo tutte quelle foto in cui compare l'oggetto di interesse che si vuole riconoscere su uno sfondo casuale che non sia in alcun modo correlato con esso. Dunque, ho provveduto a scrivere un semplice *script* in Python che, esegue il seguente comando OpenCV:

```
opencv_createsamples -img positives/image_i.jpg -bg negatives.txt
-info.image_i.txt -num 210 -maxidev 100 -maxxangle 0.0 -maxyangle 0.0
-maxzangle 0.9 -bgcolor 0 -bgthresh 0 -w 48 -h 48
```

il quale combina ogni immagine della moneta da 2€ ottenuta dallo *stream* video con 210 immagini casuali presi dall'archivio online ottenendo quindi circa 12600 nuove immagini di allenamento in cui, come nella seguente, il positivo è collocato in posizione casuale nello sfondo ed è più o meno ruotato:

Figura 3.5: Esempio di positivo utilizzato nell'algoritmo di riconoscimento di OpenCV

Naturalmente, è intuibile che più si riesca a creare immagini del genere utilizzando quanti più sfondi possibili e positivi, meglio funzionerà il processo di riconoscimento alla fine. Una volta poi ottenuta questa grande mole di immagini, che chiameremo ***training images***, dopo aver raccolto tutti i file .txt creati dall'esecuzione iterata del precedente comando in un unico file, detto `positives.txt`, e dopo aver anche creato un vettore che descriva le *positive images* col seguente comando:

```
opencv_createsamples -info positives.txt -bg negatives.txt -vec 2.vec
-num 12600 -w 48 -h 48
```

occorre lanciare il processo vero e proprio di allenamento che basandosi sulle *training images*, sulle negative e su quelle di *training* cerca di ottenere un file *classifier.xml* da usare per il riconoscimento. Questo processo, lanciato dal comando:

```
opencv_traincascade -data final -vec 2.vec -bg negatives.txt -numPos 12000
-numNeg 3000 -numStages 20 -featureType HAAR -precalcValBufSize 1024
-precalcIdxBufSize 1024 -minHitRate 0.999 -maxFalseAlarmRate 0.5 -w 48
-h 48 -mode ALL
```

Dal comando appena lanciato, possiamo notare che l'algoritmo di riconoscimento scelto ed utilizzato è l'**HAAR**. In effetti OpenCv supporta due categorie di

algoritmi di riconoscimento:

LBP cascade : molto veloce ma molto meno accurato di HAAR. Esso basa i suoi calcoli solo su numeri interi ed è l'ideale quando OpenCV deve essere utilizzato su dispositivi mobili o *embedded*. Inoltre, è preferibile utilizzare quest'algoritmo qualora di disponga di *positive images* di alta qualità;

HAAR cascade : estremamente più lento del precedente ma molto più accurato, utilizza per i calcoli i numeri a *floating points*. Poiché l'allenamento è stato effettuato su una macchina a parte (e non naturalmente sull'iPhone) ho deciso di utilizzare quest'ultimo.

Come ci si aspettava, si è rilevato che l'algoritmo HAAR richiede altissime risorse di calcolo, sia in termini di quantità di ram, sia in termini di potenza di calcolo vera e propria sia in termini di tempo. Infatti, questo procedimento non solo ha reso il server praticamente inutilizzabile durante tutto il suo percorso (saturando tutta la RAM disponibile e prendendo il totale controllo del processore) ma ha richiesto oltre una settimana per concludersi a causa dell'enorme quantità di immagini la quale, tuttavia, era pur sempre relativamente modesta in termini di OpenCV. Dunque, cercare di ottenere un `classifier.xml` che riconoscesse anche il pesce era del tutto impossibile non solo per le comuni di risorse di calcolo limitate all'utente medio ma anche perché avrebbe richiesto un numero di immagini notevole per ogni tipo di pesce il che rendeva il lavoro già di per sé estremamente lungo ed oneroso, in quanto non si trovano analoghi database di immagini online. A questo va aggiunto anche che la documentazione disponibile online su OpenCV è limitata ma soprattutto è poco chiara il che ha causato l'errata esecuzione di molti processi di allenamento che richiedevano quindi di essere ri-avviati da capo.

Al termine del processo di allenamento finale, sono riuscito ad ottenere un classificatore ed ho scritto il seguente semplice programma in C++ per testarlo con le API di OpenCV:

Frammento di codice 3.1: Esempio di semplice programma OpenCV per riconoscere la moneta da 2€

```
1
2  /* COMPILE WITH:
3  g++ -I/usr/local/include -L/usr/local/lib/ -g -o binary foto.cpp
       -L/usr/local/lib -lopencv_shape -lopencv_stitching
       -lopencv_objdetect -lopencv_superres -lopencv_videostab
       -lopencv_calib3d -lopencv_features2d -lopencv_highgui
       -lopencv_videoio -lopencv_imgcodecs -lopencv_video -lopencv_photo
       -lopencv_ml -lopencv_imgproc -lopencv_flann -lopencv_core
4  */
5
6  #include "opencv2/opencv.hpp"
7  #include <opencv2/core/ocl.hpp>
8
9  cv::Rect getBiggestOne(std::vector<cv::Rect> money) {
10
11         cv::Rect biggestRect(0,0,0,0);
12         for(size_t i = 0; i < money.size(); ++i) {
13                cv::Rect rect = money[i];
14                if(rect.width > biggestRect.width) {biggestRect = rect;}
15         }
16
17         return biggestRect;
18
19 }
20
21 using namespace cv;
22
23 int main(int, char**) {
24
25         Mat src = imread("images/2aa.jpg");
26
27         Mat src_gray;
28
```

```
29          std::vector<cv::Rect> money;
30
31          CascadeClassifier euro2_cascade;
32
33          cv::ocl::setUseOpenCL(false);
34
35          cvtColor(src, src_gray, CV_BGR2GRAY );
36          equalizeHist(src_gray, src_gray);
37
38          if ( !euro2_cascade.load( "cascade.xml" ) ) {
39          printf("--(!)Error loading\n");
40          return -1;
41          }
42
43          euro2_cascade.detectMultiScale( src_gray, money, 1.005, 0,
                0|CV_HAAR_SCALE_IMAGE, cv::Size(10, 10),cv::Size(4000, 4000)
                );
44          printf("%d\n", int(money.size()));
45
46          cv::Rect biggestRect = getBiggestOne(money);
47         cv::Point center( biggestRect.x + biggestRect.width*0.5,
              biggestRect.y + biggestRect.height*0.5 );
48         ellipse( src, center, cv::Size( biggestRect.width*0.5,
              biggestRect.height*0.5), 0, 0, 360, Scalar( 255, 0, 255 ), 4,
              8, 0 );
49
50          imwrite("result.jpg",src);
51  }
```

Questo programma è stato testato utilizzando nuovi sfondi negativi scaricati dalla rete, diversi da quelli usati per l'allenamento, nei quali è stata inserita una moneta da 2€ e questi sono alcuni dei risultati ottenuti:

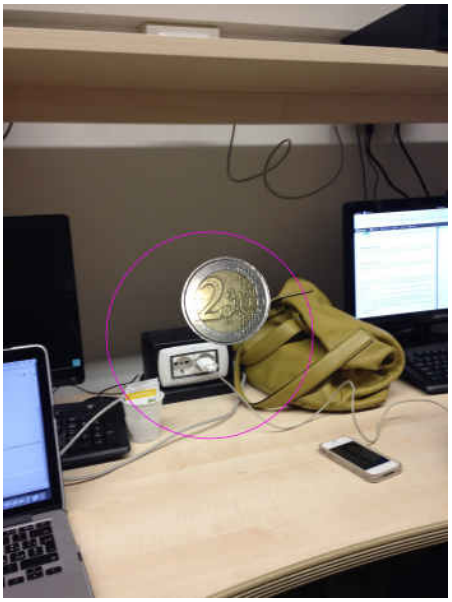
(a)

(b)

(c)

(d)

(e)



(f)

Figura 3.6: Risultati ottenuti dal tentativo di riconoscimento della moneta da 2€

Come possiamo vedere quindi i risultati ottenuti sono deludenti, sebbene alcuni siano parzialmente positivi. Possiamo comunque considerarli divisi in due gruppi:

- nelle immagini (b), (c), (d), (f) la moneta da 2€ è stata effettivamente riconosciuta. Tuttavia l'area che, secondo il programma, contiene tale riconoscimento è molto più larga rispetto a quella reale quindi, qualora anche la *pincher view* vi fosse posizionata sopra, l'utente non sarebbe comunque sollevato dal *task* oneroso di doverla ridimensionare considerevolmente;

- mentre invece, nelle immagini (a) ed (e) la moneta da 2€ non viene riconosciuta per niente. Ancora peggio, la vista viene posizionata su una porzione dell'immagine che non c'entra nulla con la moneta. Queste due immagini erano proprio quelle che ci interessavano di più in quanto propongono lo scenario tipico di uso dell'app: una foto con moneta e pesci ed una foto di una moneta appoggiata ad un tavolo, operazione che tipicamente ci aspettiamo che l'utente farà per scattare la foto. Le cause possibili di questo potrebbero essere due:

  1. cattiva qualità delle immagini positive, il che quindi richiederebbe di girare un nuovo video alla moneta da 2€ ed estrarre dei fotogrammi di qualità e condizioni di illuminazione migliori;

  2. l'uso degli sfondi usati nel processo di allenamento non sono analoghi a quelli usati nelle foto di prova. Infatti, vediamo che la moneta da 2€ viene effettivamente riconosciuta solo in quelle immagini dove lo sfondo è molto particolareggiato e complesso, che richiama alla vista quindi una vera e priopria scena; mentre, su superfici più o meno uniformi (come quella di un tavolo) la moneta non viene riconosciuta affatto il che potrebbe essere giustificabile dal fatto che l'algoritmo di *training* di OpenCV si è allenato considerando solo una certa tipologia di sfondi.

A questo punto quindi si richiederebbe di fare un nuovo tentativo di allenamento utilizzando come *negatives* delle immagini di superfici uniformi le quali siano correlate col concetto di "tavola", e quindi ci riferiamo soprattutto a superfici di

legno, marmo o anche di tovaglie. Se questo nuovo allenamento portasse a risultati migliori, anche solo il riconoscimento della moneta senza che la vista posizionatavi sopra sia necessariamente della giusta dimesione, si potrebbe considerare l'opportunità di integrare questo *framework* all'interno dell'app, liberando così l'utente almeno dall'operazione di *dragging* della giusta *pincher view* sul relativo oggetto per cui è stata implementata. Tuttavia, le probematiche maggiori sono non solo cercare un archivio molto grande di sfondi superfici ma anche un'eventuale spesa di affitto di grossissime risorse di calcolo ("macchine virtuali" basate sui decine di GB di memoria RAM e di molti processori in parallelo) affinché il processo di allenamento venga velocizzato e reso di durata minore possibile il che, ripetiamo, non è fattibile dal punto di vista dell'utente medio ma richiede l'affitto di risorse di calcolo nel *cloud* il cui pagamento spesso non è nemmeno economico. Per concludere, a dispetto del fallimento di questo tentativo, possiamo comunque assicurare che l'app rispetta in pieno i requisiti richiesti purché l'utente sia dotato delle caratteristiche esplicate nella sezione precedente.

Appendice A

Prima Appendice: le classi PHP Fish e Report

Il file `fish.php`

```
<?php
class fish {
  public $config;
  public function __construct($conf) {$this->config=$conf;}

  public function fishInfo($id){
    $sql = mysqli_query($this->config->dbConn,"SELECT * FROM
        ".$this->config->table_fishes." WHERE id='".$id."'");
        $v = mysqli_fetch_array($sql,MYSQLI_ASSOC);
        $tmp = array(
          'id'=>$v['id'],
          'commName'=>$v['commName'],
          'sciName'=>$v['sciName'],
          'minSize'=>$v['minSize'],
          'reproduction'=>$v['reproduction'],
        );
        return $tmp;
   }
```

```
18
19    public function allFishes($where) {
20      $sql = mysqli_query($this->config->dbConn,"SELECT * FROM
            ".$this->config->table_fishes." ".$where);
21          $all=array();
22          while($v = mysqli_fetch_array($sql, MYSQLI_ASSOC)){
23            $tmp = $this->fishInfo($v["id"]);
24            array_push($all, $tmp);
25          }
26          return $all;
27    }
28 }
29 ?>
```

Il file report.php

```
1  <?php
2  class report {
3    public $config;
4    public function __construct($conf) { $this->config=$conf; }
5
6    public function addReport($value) {
7      $sql = "INSERT INTO ".$this->config->table_reports." (`id_fish`,
           `fish_name`, `detectedSize`, `eligibilitySize`,
           `eligibilityDate`, `origin`, `production`, `townReport`) VALUES
           ('".$value[1]."','".$value[2]."','".$value[3]."','".$value[4]."','".$value[5].
8                                                    "','".$value[6]."','".$value[7].
9                                                    "','".$value[8]."')";
10     if (mysqli_query($this->config->dbConn,$sql)) {return;
           //mysqli_insert_id($this->config->dbConn);}
11     else {
12       //return mysqli_error($this->config->dbConn);
13       return mysqli_connect_errno($this->config->dbConn);
14     }
```

```
15    }
16 }
17 ?>
```

Appendice B

Seconda Appendice: gli script PHP che comunicano con l'app

Il file `handleFishes.php` per ottenere i dati sui pesci

```php
<?php
  require_once("config.php");
  $sistema = new config();

  if (isset($_GET["all_fishes"])) {
    if (isset($_GET["month"])) {
          $allFishes = $sistema->fish->allFishes("WHERE NOT
              (reproduction LIKE '%".$_GET["month"]."%')");
        }
        else {$allFishes = $sistema->fish->allFishes("ORDER BY id ASC");}
        echo json_encode($allFishes);
        }
?>
```

Il file `handleReport.php` per salvare i dati sul report

```php
<?php
  require_once("config.php");
  $sistema = new config();
```

```
4
5   if (isset($_POST["action"])) {
6     if ($_POST["action"] == "save_report") {
7           $value = array(null, $_POST["id_fish"], $_POST["fish_name"],
8                         $_POST["detectedSize"],
                              $_POST["eligibilitySize"],
                              $_POST["eligibilityDate"],
9                         $_POST["origin"], $_POST["production"],
                              $_POST["townReport"]);
10          echo $sistema->report->addReport($value);
11         }
12  }
13 ?>
```

N.B.: nella classe `config.php`, che non viene riportata per questioni di sicurezza, oltre che contenere i riferimenti alle altre classi in Appendice A, sono inserite anche tutte le informazioni di accesso al database come nomi delle tabelle e credenziali di acesso.

Bibliografia

[1] Vandad Nahavandipoor
iOS 8 Swift Programming Cookbook
Novembre 2014
O'Reilly Media, Inc.

[2] **StackOverflow**
`http://stackoverflow.com/`

[3] Tutorial online residenti su sito dedicato alla programmazione Apple
`http://www.raywenderlich.com/category/ios`
`http://www.raywenderlich.com/category/swift`

[4] Guide ufficiali rilasciate da Apple sulla programmazione Swift e disponibilii presso l'app iBooks
The Swift Programming Language (versioni 1.2 e 2.0)
Using Swift with Cocoa and Objective-C

[5] **iOS Developer Library**
`https://developer.apple.com/library/ios/navigation/`

[6] **Documentazione e forum ufficiali di OpenCV 2.4 (validi anche per la versione 3.1**
`http://answers.opencv.org/questions/`
`http://docs.opencv.org/2.4/`
`http://docs.opencv.org/3.1.0/`

[7] **Tutorial sulla creazione di file classifier con OpenCV**
http://www.memememememememe.me/training-haar-cascades/
http://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html

[8] **CENSIS-COLDIRETTI**
Primo rapporto sulle abitudini alimentari degli italiani. Sintesi dei principali risultati.
Roma: 27 pp, (2010)

[9] **FAO - ISMEA (2007)**
Il settore ittico in Italia e nel mondo: le tendenze recenti.
Roma: 376 pp, (2011)
Food Outlook www.fao.org

[10] **ISMEA**
I consumi ittici nei principali Paesi europei. Roma: 160 pp., 2008
Acquacoltura. Report economico-finanziario. Roma: 128 pp., 2009
Compendio statistico del settore ittico. Roma: 96 pp., 2009
Il settore ittico in Italia. Check up ittico 2009. Roma: 48 pp., 2010
Il settore ittico in Italia. Check up ittico 2010. Roma: 52 pp., 2011
Il pesce a tavola: percezioni e stili di consumo degli italiani, 24 pp.
Tendenze ittico. Trimestrale di analisi e previsioni per i settori agroalimentari.
www.ismea.it

[11] **ISTAT**
Rapporto annuale. La situazione del Paese nel 2010: 438 pp., 2011

[12] **OECD-FAO**
Agricultural Outlook 2011-2020.
www.agri-outlook.org

Ringraziamenti

Un grazie sentito al mio relatore **Luciano Bononi**, Professore associato del Dipartimento di Informatica della Scuola di Scienze dell'Università degli Studi di Bologna, per avermi dato la possibilità di continuare e coltivare maggiormente il mio interesse nelle app del mondo Apple.

Ringrazio anche la Dott.ssa **Flavia Bartoccioni** del Corso di Dottorato in Biologia Evoluzionistica ed Ecologia della Facoltà di Scienze di Tor Vergata, coordinata dalla Professoressa **Caterina Lorenzi** del Laboratorio di Didattica di Ecologia. Grazie per avermi dato la possibilità di realizzare un'app originale e per la vostra presenza ed aiuto durante tutto il mio percorso.

Ultimo, ma non meno importante, un grazie di cuore a tutte le persone, soprattutto gli amici e compagni di corso, che si sono prestati come utenti per il testing di tutte e tre la fasi della mia app discusse in questo lavoro:

Mauro Maestri
Claudia Carpineti
Federico Barocci
Luca Pompei
Antonello Antonacci
Pierpaolo Del Coco
Giuseppe Turlione
I miei genitori, Laura e Giovanni, e mio fratello Francesco
Cecilia Roda
Luca La Sala
Ivan Heibi