

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра дифференциальных уравнений и системного анализа

**РАЗРАБОТКА И РЕАЛИЗАЦИЯ МОБИЛЬНОГО ПРИЛОЖЕНИЯ
ДЛЯ ОЦЕНКИ КРАСОТЫ ЧЕЛОВЕЧЕСКИХ ЛИЦ ПО
ФОТОГРАФИИ НА ЯЗЫКЕ PYTHON**

Курсовая работа

Клименко Кирилла
Владимировича

студента 3 курса,
специальность 1-31 03
09 Компьютерная
математика
и системный анализ

Научный
руководитель:
Воробей В. А.

Минск, 2021

ОГЛАВЛЕНИЕ

Введение	1
Глава 1. Подготовка набора данных	3
1.1 Знакомство с набором данных	3
1.2 Анализ данных	4
1.3 Генерация обучающей и тестовой выборки	7
Глава 2. Обучение нейросетевой модели	8
2.1 Постановка задачи	8
2.2 Сравнение некоторых нейросетевых моделей	8
2.3 Подготовка и обучение нейросетевой модели	10
2.4 Распознавание границы и частей лица на фотографии	14
2.5 Демонстрация результатов	17
Глава 3. Разработка и развёртывание веб-клиента	18
3.1 Постановка задачи	18
3.2 Разработка веб-приложения	20
3.3 Развертывание веб-приложения	22
Глава 4. Проектирование и реализация мобильного приложения	24
4.1 Постановка задачи	24
4.2 Обработка и анализ лицевых параметров	25
4.3 Проектирование основного интерфейса приложения	28
4.4 Реализация основной логики приложения	34
4.5 Сборка приложения под платформу Android	36
Заключение	39
Список использованной литературы	40
Аннотация к репозиторию проекта	41

ВВЕДЕНИЕ

Ещё пару десятилетий назад представления о том, как понять, что такое красота человека, были достаточно примитивными. Рассуждения о том, какое лицо считается красивым с точки зрения математики, сводились к тому, что оно должно удовлетворять законам симметрии. Также со времен эпохи Возрождения были попытки описать красивые лица при помощи соотношений между расстояниями в каких-то точках на лице и показать, например, что у красивых лиц какое-то отношение близко к золотому сечению. Подобные идеи о расположении точек сейчас используются как один из способов идентификации лиц.

Так можно ли автоматизировать оценку человеческой красоты? Эта цель может поднять дюжину интересных философских и этических вопросов. В какой мере понятие красоты обусловлено расовой принадлежностью и культурой? Насколько верно, что каждый индивидуален в своих предпочтениях? Возможно ли вообще выделить объективные признаки красоты? В поисках ответов на эти и не только вопросы, в рамках данной работы будет представлен анализ статистических данных об оценках красоты одних людей другими, а также процесс проектирования и обучения нейросетевой модели, оценивающей красоту человеческого лица по фотографии на базе веб-клиента, выступающего в качестве вспомогательного ресурса обработки поточных запросов, и мобильного приложения под управлением операционной системы Android, использующегося в качестве удобного пользовательского интерфейса. Полная документация и реализация проекта, получившего название BeautyDeer, будет доступна в [GitHub-репозитории \[1\]](#) под открытым лицензированием GNU GPL v3.0.

ГЛАВА 1. ПОДГОТОВКА НАБОРА ДАННЫХ

1.1 Знакомство с набором данных

Поскольку внешность, как и личностные черты, достаточно личная и деликатная тема для каждой персоны, в публичном доступе крайне мало наборов данных, содержащих фотографии человеческих лиц с оценкой их красоты. На Рисунке 1.1 представлены все датасеты для прогнозирования красоты лица на период до 2020 года.

Database	Image Num.	Labelers/Image	Beauty Class	Face Property	Face Landmarks	Publicly Available
Y. Eiseenthal et al. [15]	184	28 or 18	7	Caucasian Female	×	×
F. Chen et al. [22]	23412	unknown	2	Asian Male/Female	✓	×
H. Gunes et al. [35]	215	46	10	Female	✓	×
J. Fan et al. [23]	432	30	7	Generated Female	✓	×
M. Redi et al. [34]	10141	78-549	10	Multiple (Sampled from AVA [16])	×	✓
SCUT-FBP [1]	500	70	5	Asian Female	✓	✓
SCUT-FBP5500	5500	60	5	Asian/Caucasian; Male/Female	✓	✓

Рис. 1.1 Датасеты для прогнозирования красоты [2]

Для поставленной задачи предсказания красоты воспользуемся базой данных SCUT-FBP5500, собранной одним китайским университетом, которая содержит 5500 фотографий нейтральных, фронтально расположенных лиц, возрастом от 15 до 60 лет, каждая из которых оценена множеством независимых разметчиков с помощью специально разработанной университетом системой маркировки на базе Ali Cloud. Из этого числа фотографий:

- 2000 – азиатские мужчины (AM);
- 2000 – азиатские женщины (AF);
- 750 – европеоидные мужчины (CM);
- 750 – европеоидные женщины (CF).

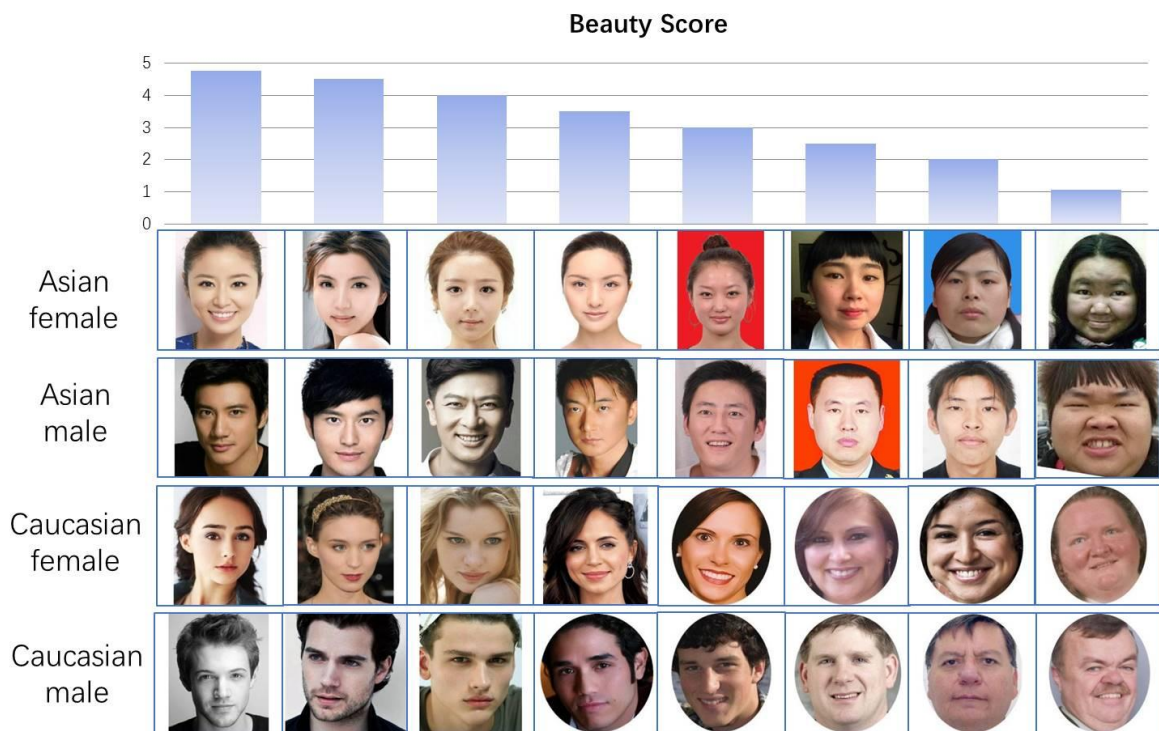


Рис. 1.2 SCUT-FBP5500 [2]

Все изображения датасета SCUT-FBP5500 имеют оценки красоты от 1 до 5, где оценка красоты 5 означает самое привлекательное лицо.

Более подробную информацию об используемом датасете можно почитать в официальном документе “Diverse Benchmark Dataset for Multi-Paradigm Facial Beauty Prediction” [2].

1.2 Анализ данных

С помощью модуля Pandas из Python посмотрим на набор данных с различных сторон.

Исходя из Рисунка 1.3 можно сказать, что в целом:

- 1) Женщин считают более привлекательными, чем мужчин;
- 2) Азиатские женщины привлекательнее, чем европеоидные;
- 3) Европеоидные мужчины привлекательнее, чем азиатские;
- 4) Большинство женщин и мужчин имеют средние показатели привлекательности;
- 5) Низких оценок привлекательности почти нет, поэтому позже мы немного перенормируем данные для более наглядного результата.

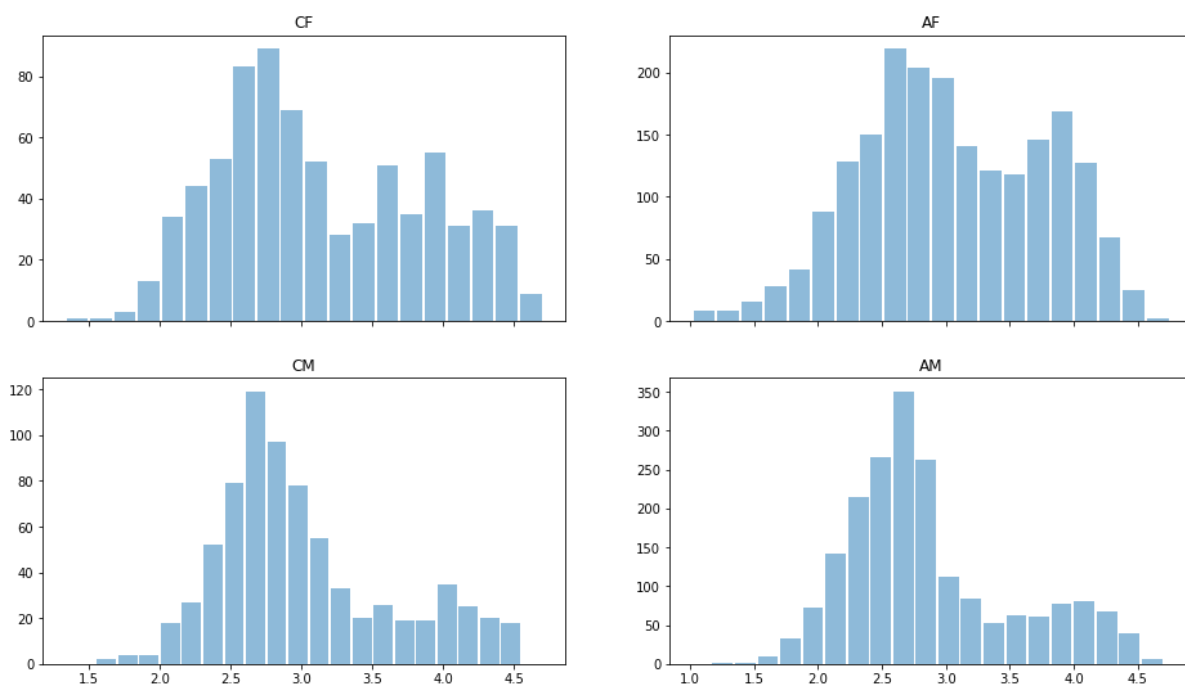


Рис. 1.3 Распределение оценок для различных полов и рас

Посмотрим среднеквадратичное отклонение: оно составляет 0.6431333, что говорит об отсутствии индивидуальности в предпочтениях людей, так как разница между оценщиками составляет менее одного балла из пяти.

Говоря о единодушии в оценивании красоты, можно сказать, что мнения только 2.9% оценщиков отличаются более, чем на один балл. Схожесть в виденьях красоты можно проиллюстрировать, построив распределение отклонений оценок от их медианного значения (см. Рис. 1.4) и коэффициентов корреляции для разметчиков мужского и женского пола (см. Рис. 1.5).

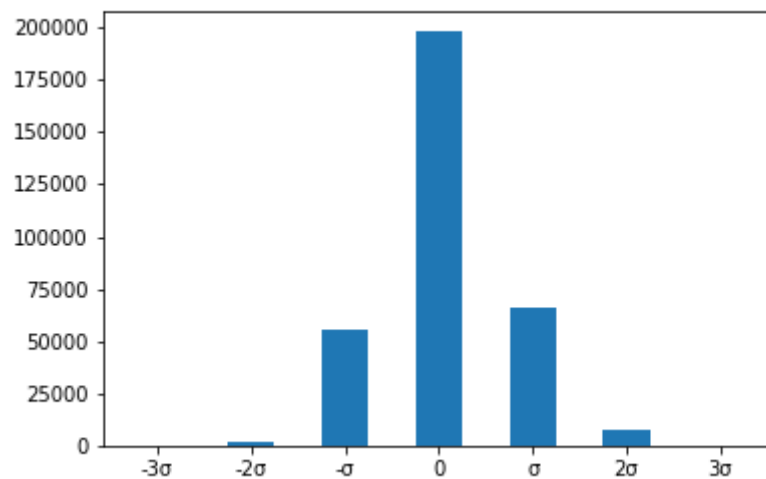


Рис. 1.4 Распределение отклонения оценки от медианной

	CF	AF	CM	AM	All Faces
Female Labelers	0.785	0.800	0.747	0.793	0.785
Male Labelers	0.791	0.795	0.763	0.797	0.781
All Labelers	0.788	0.785	0.743	0.782	0.770

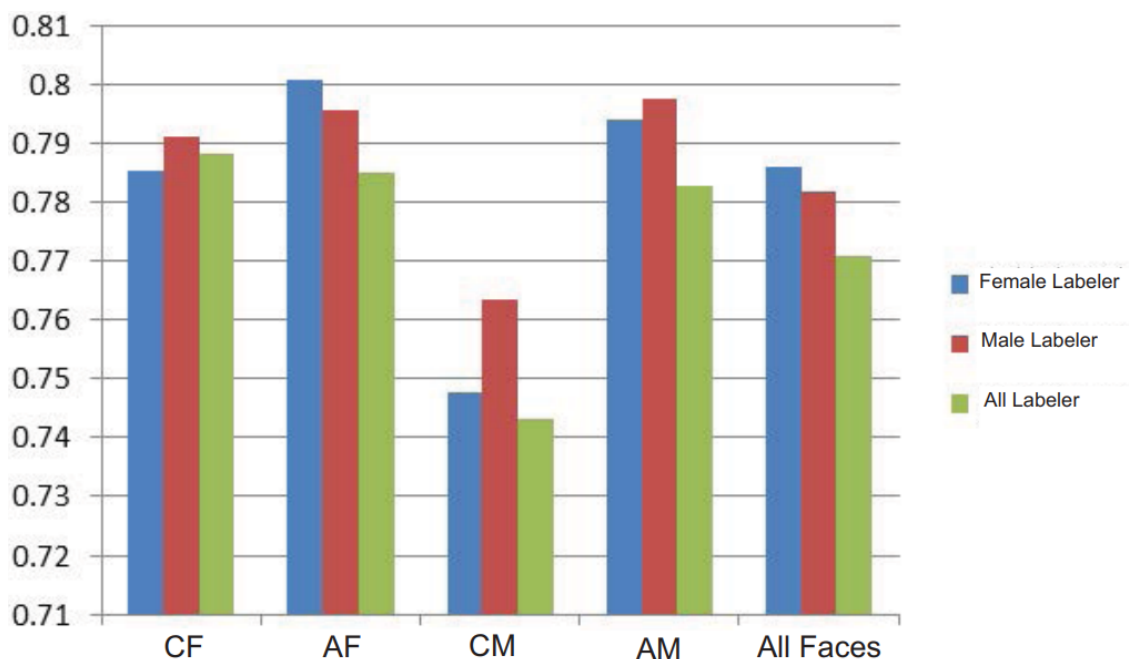


Рис. 1.5 Коэффициенты корреляции для разметчиков разных полов

Из проведённого анализа можно сделать вывод, что набор данных SCUT-FBP5500 хорошо подойдёт для обучения нейросетевой модели, оценивающей привлекательность человеческих лиц.

1.3 Генерация обучающей и тестовой выборки

Одним из наиболее влияющих на точность нейросетевой модели факторов является обучающая выборка. Поэтому, в угоду качеству нашей работы, извлечем данные из датасета (см. [\[1\]/neural-network/data_extraction.py](#)) и проведём над ними следующие действия:

- 1) Очистка данных – из каждой фотографии, представленной в датасете, возьмём только лицо. Для этого воспользуемся библиотекой компьютерного зрения OpenCV [\[3\]](#), с помощью которой найдём границы лица и обрежем изображение по этим границам (см. [\[1\]/neural-network/generate_data.py/detect_face](#)).
- 2) Обогащение данных – для каждой фотографии, представленной в датасете, осуществим аугментацию путём случайного поворота, изменения яркости, контраста и основных цветов (см. [\[1\]/neural-network/generate_data.py/random_update](#)).
- 3) Агрегирование данных – для каждой фотографии, представленной в датасете, найдём среднюю оценку красоты из всех оценок данной фотографии, сделанных различными разметчиками (см. [\[1\]/neural-network/generate_data.py/get_lable_distribution](#)).
- 4) Смешение и разделение данных – случайным образом перемешаем получившиеся данные и разделим их на два подмножества: 90% – обучающее и 10% – тестовое (см. [\[1\]/neural-network/generate_data.py/split_data](#)).

В результате вышеперечисленных процедур мы получим готовые обучающую и тестовую выборки, на которых уже можно тренировать и тестировать нейросетевую модель. Сохраним их в соответствующие бинарные “train_lable_distribution.dat” и “test_lable_distribution.dat” файлы, которые будут содержать нормированное фото лица и его среднюю оценку привлекательности по каждому из разметчиков.

ГЛАВА 2. ОБУЧЕНИЕ НЕЙРОСЕТЕВОЙ МОДЕЛИ

2.1 Постановка задачи

Чтобы определить красоту человеческого лица по фотографии как можно точнее, мы должны посмотреть в сторону готовых архитектурных решений глубоких нейронных сетей, которые за эти годы научились довольно успешно решать подобные задачи, и выбрать из них наиболее подходящую и точную модель. Кроме того, выбранная нами базовая архитектура должна быть достаточно лёгкой и быстрой для того, чтобы эффективно обрабатывать поток фотографий и не потреблять при этом большое количество ресурсов.

После того, как мы определимся с базовой моделью, мы дообучим её на подготовленном ранее наборе данных, заморозив свёрточные слои и добавив новые, а также поэкспериментируем с различными настройками и параметрами для улучшения качества предсказания новой модели.

Как только модель будет обучена и готова к использованию, мы позаботимся о качественном распознавании человеческого лица на любой фотографии, чтобы впоследствии подать его на вход нейросети в нужном формате.

2.2 Сравнение некоторых нейросетевых моделей

По результатам исследования [\[4\]](#) команды инженеров по машинному обучению из Китая, наиболее успешными в задаче классификации человеческих лиц по заданными меткам привлекательности оказались модели AlexNet, ResNet-18, ResNet-50, ResNet-101 и VGGNet-19.

Сравним данные модели на датасете SCUT-FBP5500 [\[2\]](#) по следующим метрикам (см. Рис. 2.1):

- корреляция Пирсона (PC);
- средняя абсолютная ошибка (MAE);
- среднеквадратичная ошибка (RMSE).

$$PC = \frac{\sum_{i=1}^m (\hat{l}'_i - \bar{\hat{l}}')(l'_i - \bar{l}')}{\sqrt{\sum_{i=1}^m (\hat{l}'_i - \bar{\hat{l}}')^2} \sqrt{\sum_{i=1}^m (l'_i - \bar{l}')^2}}$$

$$MAE = \frac{1}{m} \sum_{i=1}^m |l'_i - \hat{l}'_i|$$

$$RMSE = \sqrt{\frac{1}{m} \sum_{i=1}^m (l'_i - \hat{l}'_i)^2}$$

Рис. 2.1 Метрики PC, MAE и RMSE

Высокие значения PC и низкие MAE и RMSE будут означать наиболее точные прогнозы оценки привлекательности лиц. Изобразив прогнозы графически, по корреляции Пирсона мы лучше сможем понять, насколько близки показатели предсказания красоты лиц обученной нейросети к результатам реальных оценщиков. Чем выше будет корреляция между прогнозами и реальными оценками, тем ближе окажется нейросетевая модель к людям в отношении понимания красоты.

Как видно из Рисунка 2.2, 50-слойный ResNet оказался лучше всех представленных моделей, протестированных на нашем наборе данных.

Метрика	AlexNet	ResNet-18	ResNet-50	ResNet-101	VGGNet-19
PC	0.8298	0.8513	0.8858	0.8763	0.8510
MAE	0.2938	0.2818	0.2871	0.2919	0.3233
RMSE	0.3819	0.3703	0.3643	0.3748	0.4296

Рис. 2.2 Сравнение моделей по метрикам PC, MAE и RMSE

Немного худшая производительность AlexNet, ResNet-101 и VGGNet-19, скорее всего, может быть обусловлена чрезмерно сложными архитектурами. Действительно, архитектуры этих моделей больше и тяжелее, по сравнению с архитектурами моделей ResNet (см. Рис. 2.3).

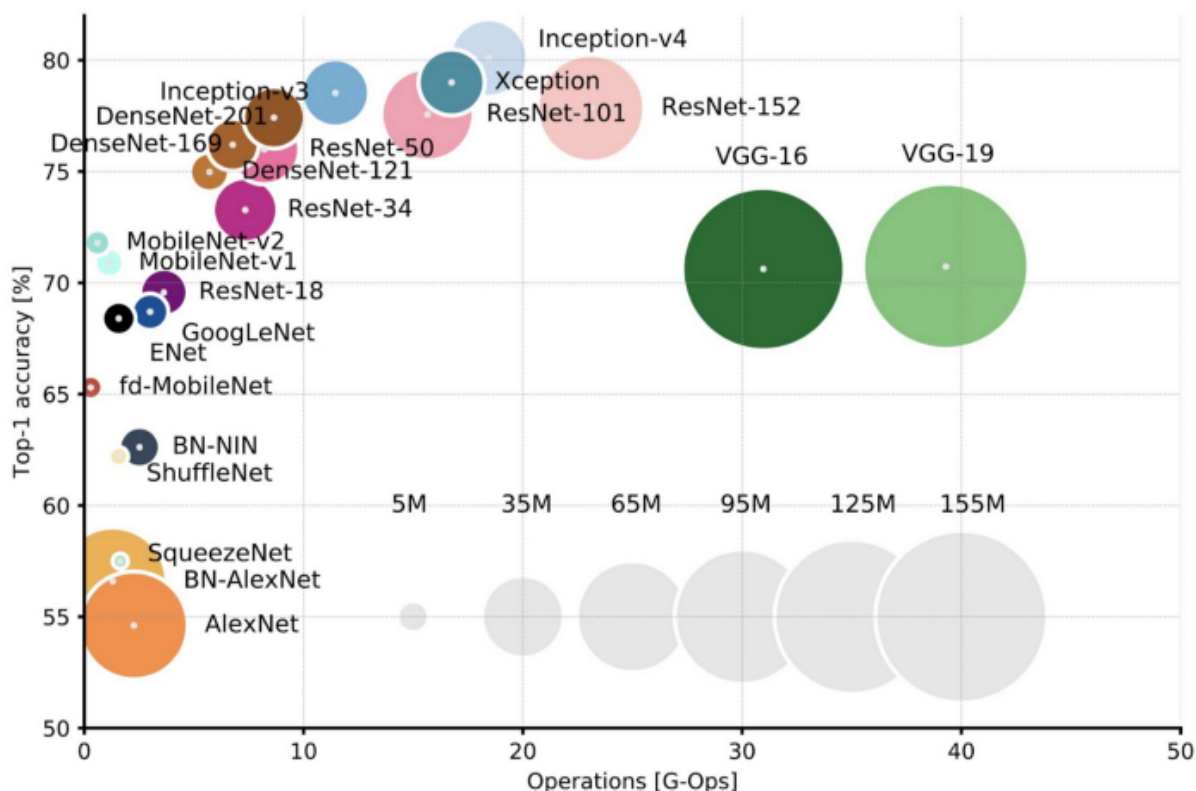


Рис. 2.3 Распределение нейросетевых моделей по размеру и точности

Исходя из результатов сравнения, для нашей задачи хорошо подойдёт модель ResNet-50, так как она показала наилучшие результаты в предсказаниях и в тоже время является производительной и легковесной, что определённо пойдёт в плюс клиентской и серверной части проекта.

2.3 Подготовка и обучение нейросетевой модели

Базовую архитектуру ResNet-50 (см. Рис. 2.4) возьмём из библиотеки для машинного обучения Keras [5]. С помощью функционала этой библиотеки мы будем обучать и тестировать нашу модель, а также в дальнейшем оценивать привлекательность лица по заданной фотографии.

С полным скриптом обучения и тестирования можно ознакомиться в соответствующем каталоге репозитория [\[1/neural-network/train.py|test.py\]](#).

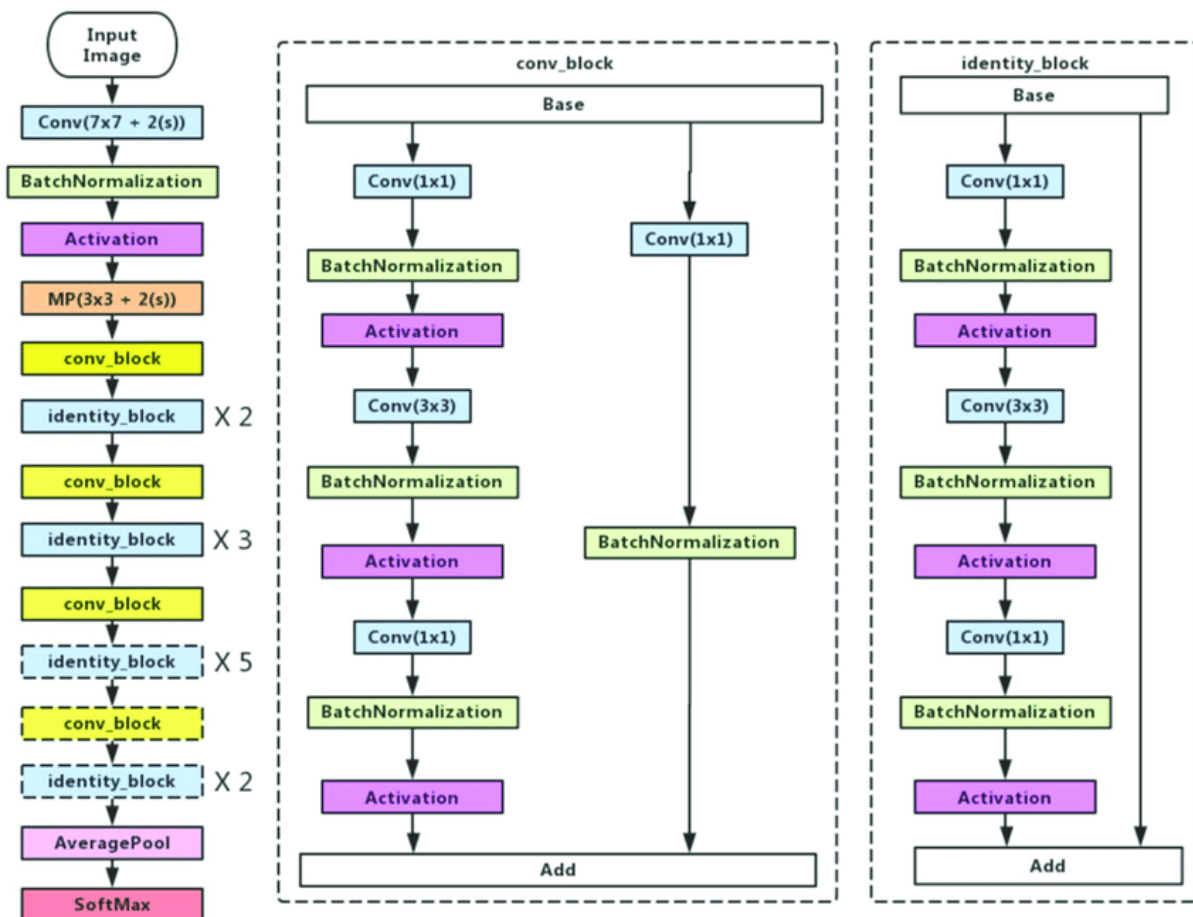


Рис. 2.4 Архитектура модели ResNet-50

Импортируем все необходимые для работы с данными и обучения нейросетевой модели модули:

```
import numpy as np
from pickle import load
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.applications.resnet50 import ResNet50
from keras.optimizers import SGD
from keras.callbacks import EarlyStopping
```

Загрузим бинарный файл с тренировочной выборкой и отделим входные параметры (изображения лиц, представленные в виде numpy-массивов) от целевой переменной (оценки красоты, представленные в виде вектора принадлежности одной из пяти оценок):

```
lable_distribution = load(open('train_lable_distribution.dat', 'rb'))
train_X = np.array([x[1] for x in lable_distribution])
train_Y = np.array([x[2] for x in lable_distribution])
```

Инициализируем модель ResNet-50, определив параметр пуллинга так, чтобы он был применён к выходным данным последнего сверточного слоя (двухмерный тензор на выходе), заморозив свёрточные слои и добавив в качестве выходного слоя пятинейронный полносвязный слой с функцией активации softmax. Таким образом, откликом последнего слоя получившейся архитектуры будет вектор вероятности принадлежности изображения к одной из пяти оценок привлекательности:

```
resnet = ResNet50(include_top=False, pooling='avg')
model = Sequential()
model.add(resnet)
model.add(Dropout(0.5))
model.add(Dense(5, activation='softmax'))
model.layers[0].trainable = False
print(model.summary())
```

Обучим полученную модель методом стохастического градиентного спуска несколько раз с разными параметрами обучения, используя в качестве функции потерь относительную энтропию и в качестве метрики качества – точность классификации. Подберём наиболее оптимальные параметры с точки зрения наименьших потерь и наилучшего качества модели, а также сохраним получившиеся результаты весов в соответствующем файле “beauty-deep-model.h5”:

```
sgd = SGD(lr=0.001, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='kld', optimizer=sgd, metrics=['accuracy'])
earlyStopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    verbose=0,
    mode='auto'
)
history = model.fit(
    x=train_X,
    y=train_Y,
    batch_size=32,
    callbacks=[earlyStopping],
    epochs=100,
    verbose=1,
    validation_split=0.1
)
model.save_weights('beauty-deep-model.h5')
```

На Рисунке 2.5 проиллюстрированы кривые обучения нейросетевой модели с наиболее оптимальными параметрами.

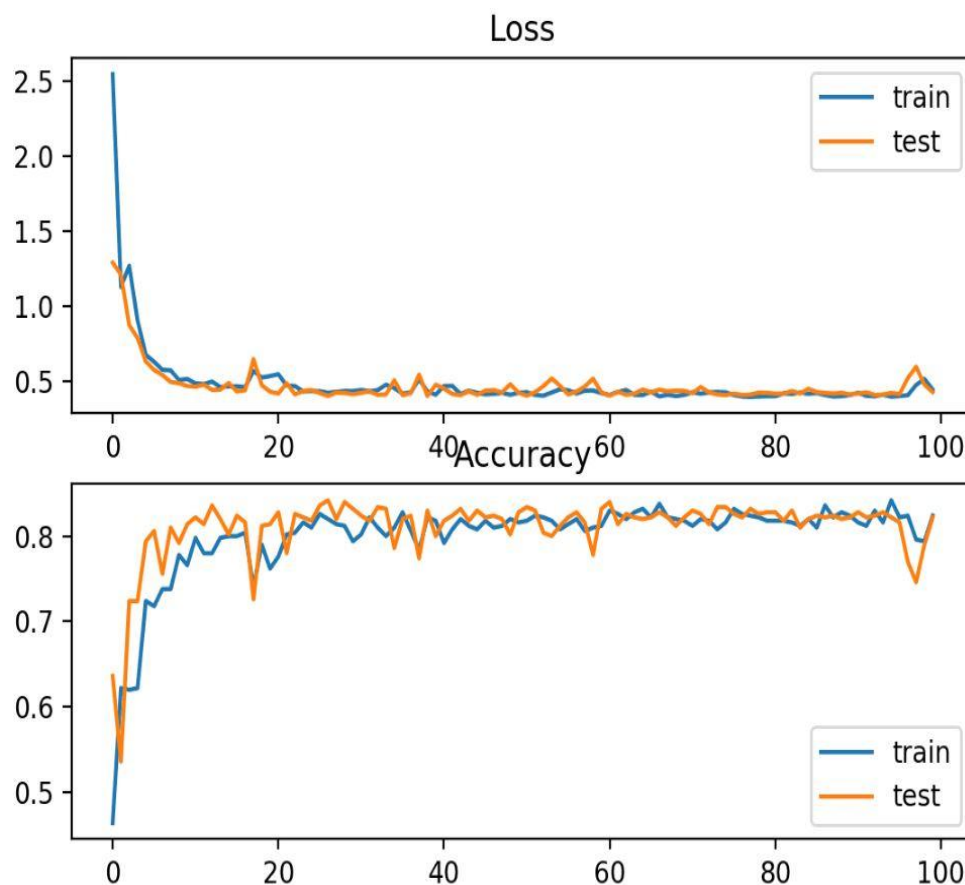


Рисунок 2.5 Кривые точности и потерь при обучении модели BeautyDeep

Вычислим корреляцию Пирсона (PC), среднюю абсолютную (MAE) и среднеквадратичную (RMSE) ошибку обученной нами модели на тестовой выборке и сравним результаты с другими моделями (см. Рис 2.6):

Метрика	AlexNet	ResNet-18	ResNet-50	ResNet-101	VGGNet-19	BeautyDeep
PC	0.8298	0.8513	0.8858	0.8763	0.8510	0.9501
MAE	0.2938	0.2818	0.2871	0.2919	0.3233	0.1485
RMSE	0.3819	0.3703	0.3643	0.3748	0.4296	0.1505

Рис. 2.6 Сравнение моделей по метрикам PC, MAE и RMSE

Исходя из результатов, проиллюстрированных на Рисунке 2.6, можно утверждать, что дообученная с некоторыми новшествами модель BeautyDeep показала себя намного лучше всех предыдущих протестированных моделей.

Изобразим графически и сравним прогнозы нейросети с реальными оценками красоты (см. Рис. 2.7):

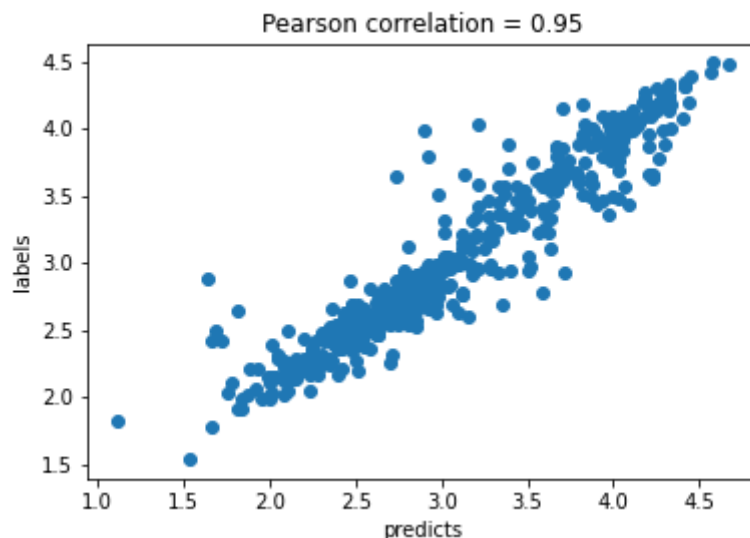


Рис. 2.7 Корреляция Пирсона модели BeautyDeer

На Рисунке 2.7 видна сильная корреляция между предсказаниями нейронной сети и оценками красоты реальных людей, что говорит о том, что модель довольно успешно смогла найти и понять некоторые законы красоты человеческих лиц разных национальностей и возрастов.

2.4 Распознавание границы и частей лица на фотографии

Для того, чтобы мы могли использовать обученную ранее нейросетевую модель на любых фотографиях, нам необходимо найти на них человеческие лица. Данная задача давно успешно реализована в компонентах детекторов библиотеки с открытым исходным кодом Dlib [\[6\]](#). С помощью этой библиотеки мы дополнительно будем собирать данные о расположении частей лица для дальнейшего анализа.

Чтобы распознать человеческие лица на фотографии, воспользуемся детектором `dlib.cnn_face_detection_model_v1`, который даст нам список объектов всех найденных им лиц:

```
detector = dlib.cnn_face_detection_model_v1(detector_path)
faces = detector(img, 0)
```

Далее, итерируясь по этому списку, обрежем каждое найденное лицо по его границе, трансформируем его и вычислим вектор вероятности оценок привлекательности с помощью нейросети:

```
for idx, f in enumerate(faces):
    face = [
        f.rect.left(),
        f.rect.top(),
        f.rect.right(),
        f.rect.bottom()
    ]
    cropped_image = img[face[1]:face[3], face[0]:face[2], :]
    resized_image = cv2.resize(cropped_image, (224, 224))
    normed_image = np.array([(resized_image - 127.5) / 127.5])
    probability_vec = model.predict(normed_image)[0]
```

Вычислим из полученного вектора точное значение оценки привлекательности и перемасштабируем её из интервала $[1;5]$ в интервал $[0;100]$, чтобы получить индекс красоты в процентах, тем самым сделав шкалу оценивания более универсальной и понятной.

Другими словами: создадим дополнительный выходной слой нейронной сети на базе обыкновенного персептрона и настроим его вручную на наше усмотрение. Входом этого слоя будет вектор вероятности принадлежности одной из оценок от 1 до 5, функцией активации будет функция масштабирования оценки в интервал $[0;100]$, а выходом – процент привлекательности лица (см. Рис. 2.8).

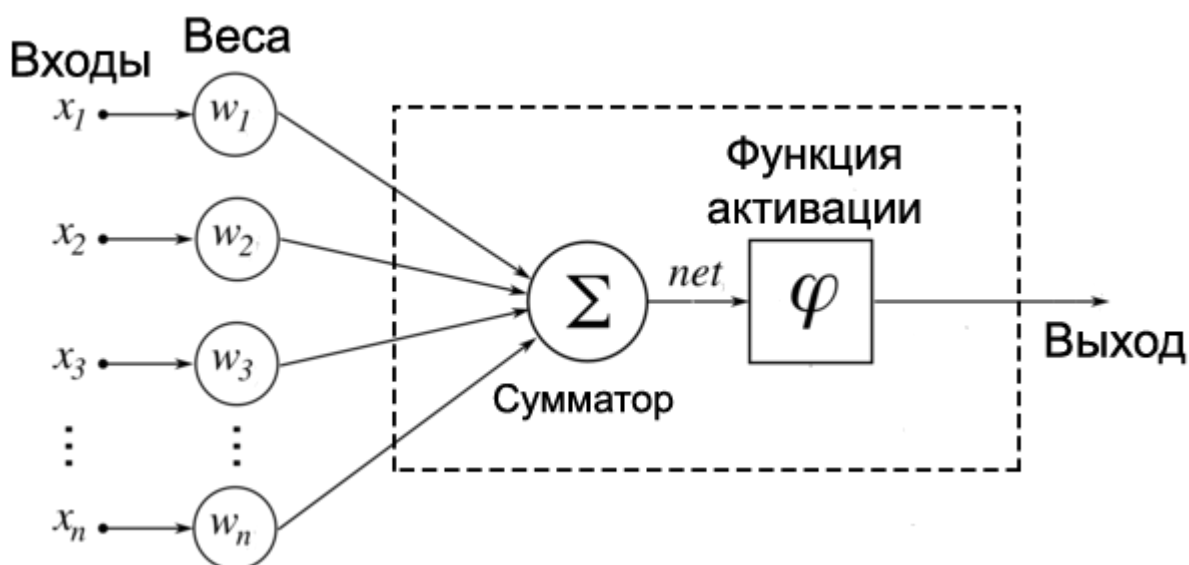


Рис. 2.8 Модель персептрона

Вычислим веса, решив простую систему уравнений, приняв, что если вероятность двух соседних оценок равна 50%, то итоговая оценка будет равняться их среднему значению:

$$\begin{cases} 0.5w_1 + 0.5w_2 + 0w_3 + 0w_4 + 0w_5 = 1.5 \\ 0w_1 + 0.5w_2 + 0.5w_3 + 0w_4 + 0w_5 = 2.5 \\ 0w_1 + 0w_2 + 0.5w_3 + 0.5w_4 + 0w_5 = 3.5 \\ 0w_1 + 0w_2 + 0w_3 + 0.5w_4 + 0.5w_5 = 4.5 \\ 0.1w_1 + 0.2w_2 + 0.4w_3 + 0.2w_4 + 0.1w_5 = 3 \end{cases} \Rightarrow \begin{cases} w_1 = 1 \\ w_2 = 2 \\ w_3 = 3 \\ w_4 = 4 \\ w_5 = 5 \end{cases}$$

В качестве функции активации возьмем функцию, переводящую нашу оценку в процентное значение (функцию можно будет в любой момент подкорректировать для получения необходимых результатов, поэтому мы и настроили этот слой вручную):

$$\varphi(x):[1;5] \rightarrow [0;100] | \varphi(x) = 25(x-1).$$

В итоге можем вычислить процент привлекательности лица, скалярно перемножив вектор вероятности на вычисленный выше вектор весов и применив функцию активации:

```
beauty_score = 25 * (np.dot(probability_vec, range(1, 6)) - 1)
```

Расположение частей лица на фотографии можно получить с помощью ещё одного детектора библиотеки Dlib – `dlib.shape_predictor`:

```
predictor = dlib.shape_predictor(predictor_path)
shape = predictor(img, box.rect)
points = [(shape.part(i).x, shape.part(i).y) for i in range(68)]
```

Данный детектор распознаёт 68 ключевых точек лица (см. Рис. 2.9), по которым можно определить, например, форму подбородка, длину носа, ширину век, угол бровей, поворот лица, наличие улыбки, значение золотого сечения и другие параметры, которые будут рассмотрены в последующих главах.

Также, согласно такой науке как физиогномика, исходя из анализа внешних черт лица и его выражения, можно определить тип личности человека, его душевные качества и даже состояние здоровья. Однако некоторые представители экспериментальной психологии относят физиогномику к числу псевдонаук. Поэтому в рамках данной работы мы не будем углубляться в тонкости влияния параметров человеческого лица на личностные качества самого человека, но это было бы интересно.

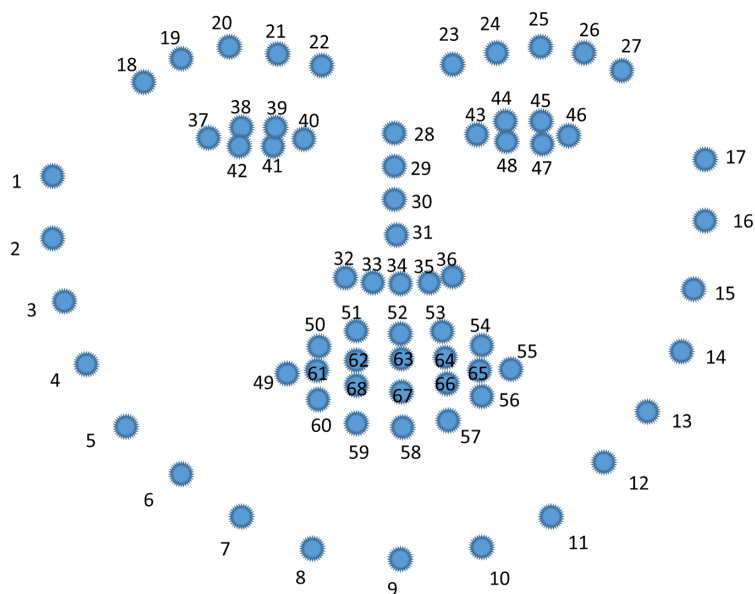


Рис. 2.9 Ключевые точки лица из `dlib.shape_predictor`

2.5 Демонстрация результатов

Объединив все описанные выше компоненты в единую технологию (или модуль) BeautyDeep [\[1/web-server/beauty_deep_net.py\]](http://web-server/beauty_deep_net.py), протестируем её на фотографиях с одним и несколькими лицами (см. Рис. 2.10).



Рис. 2.10 Результаты работы нейронной сети и детекторов

Все лица и их части на фотографиях распознаны верно. Процент привлекательности лица Клименко Кирилла (фото слева) составил 68.13%, процент красоты лиц Кита Харингтона и Эмилии Кларк (фото справа) составил 67.71% и 89.12% соответственно.

ГЛАВА 3. РАЗРАБОТКА И РАЗВЁРТЫВАНИЕ ВЕБ-КЛИЕНТА

3.1 Постановка задачи

В предыдущей главе нами реализован внушительный функционал, позволяющий распознавать человеческие лица и их части, а также оценивать красоту человека по предоставленной фотографии. Однако его настройка и использование является довольно трудоёмкой задачей для обычного пользователя, не имеющего отношения непосредственно к разработке проекта. Ведь чтобы получить необходимые пользователю результаты, ему нужно скачать все файлы и пакеты, настроить виртуальное окружение, установить все зависимости проекта, запустить встроенные архитектурные компоненты и функции в нужном порядке, предварительно разобравшись в них – и это ещё малая часть того, с чем предстоит столкнуться, не говоря об осмыслении полученных результатов.

В этой и следующей главах мы рассмотрим построение сервисной инфраструктуры по анализу красоты и других параметров человеческого лица, использующей разработанную в предыдущей главе технологию BeautyDeer, на базе веб-клиента, выступающего в качестве вспомогательного ресурса обработки поточных запросов, и мобильного приложения, использующегося в качестве удобного пользовательского интерфейса. Данный сервис упростит использование нашей технологии до простой загрузки фотографии посредством смартфона, а также обеспечит понятность и быстроту обработки результатов анализа лиц, опцию многопользовательности, мультиязычности и другие дополнительные возможности на уровне взаимодействия с пользователями, масштабируемости функционала и дизайна.

Спроектируем и опишем инфраструктуру. Она будет состоять из двух основных компонентов, обеспечивающих основу функционирования сервиса, – мобильного приложения как клиентской части и веб-приложения как серверной части (см. Рис. 3.1). Процесс обработки изображения будет описываться следующей цепочкой подпроцессов:

- 1) Загрузка изображения из файловой системы или камеры смартфона в интерфейс мобильного приложения;
- 2) Трансформация изображения до необходимого размера;

- 3) Упаковка изображения и его отправка на удалённый сервер веб-клиенту посредством REST API request;
- 4) Распаковка полученного веб-клиентом изображения;
- 5) Распознавание человеческих лиц и их частей на изображении;
- 6) Вычисление привлекательности каждого найденного на изображении лица посредством нейронной сети;
- 7) Обработка вычисленных оценок красоты и структурирование данных о лицах и их частях, найденных на изображении;
- 8) Упаковка обработанных данных о лицах в формат JSON и их отправка обратно мобильному приложению посредством REST API response;
- 9) Вычисление характеристических параметров лиц и их анализ на основе полученных мобильным приложением данных;
- 10) Отображение полученных результатов в соответствующем окне пользовательского интерфейса мобильного приложения.

Таким образом, пользователь сможет получить результаты анализа лиц сразу же после загрузки фотографии в свой смартфон.

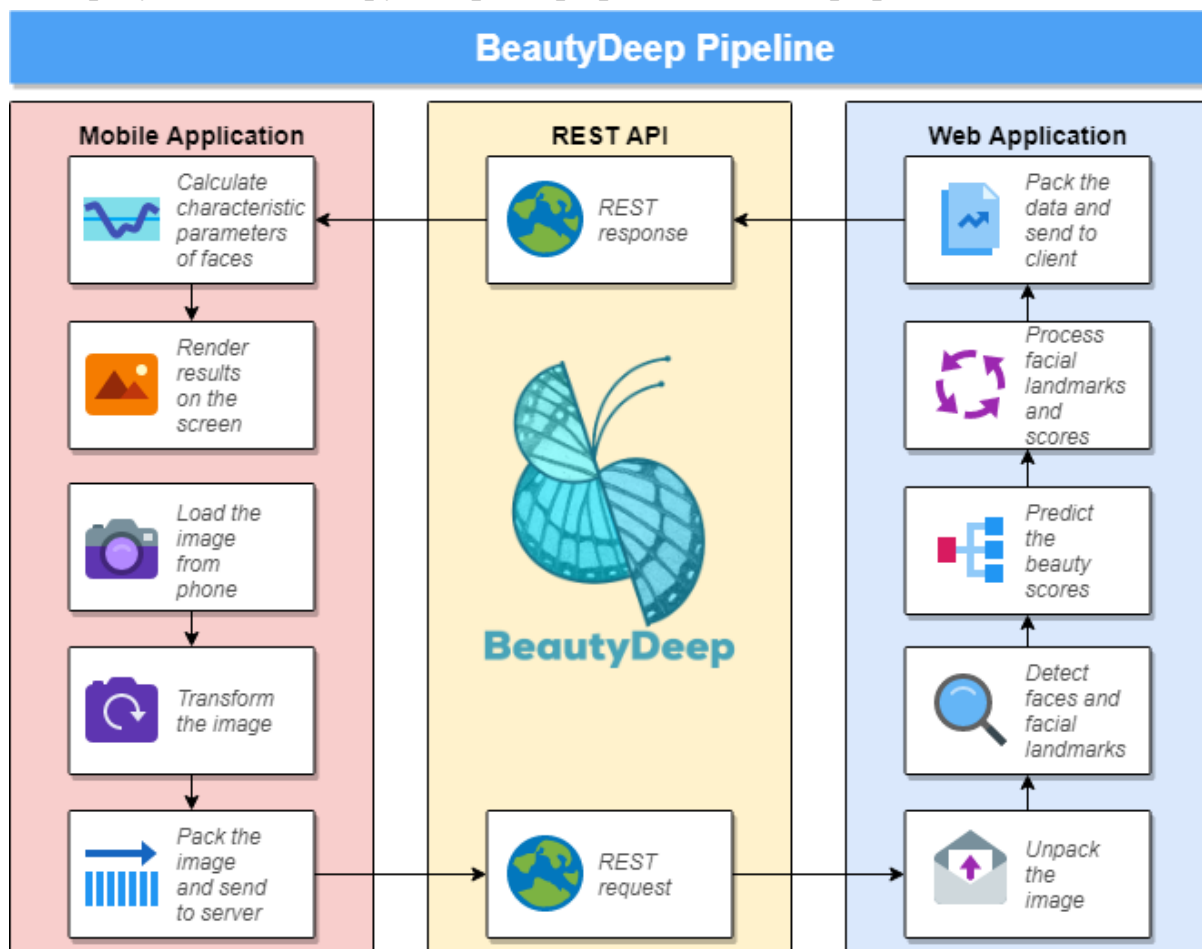


Рис. 3.1 Диаграмма инфраструктуры BeautyDeep

3.2 Разработка веб-приложения

Перед тем, как разрабатывать веб-приложение на Python, многие задаются вопросом: “Какой веб-фреймворк лучше использовать?”. Согласно данным опроса разработчиков Python в 2020, такие фреймворки как Flask [7] и Django [8] являются самыми популярными среди веб-разработчиков во всём мире.

Так как нам нужно создать быстрое, легковесное, легко масштабируемое и легко разворачиваемое веб-приложение, то наилучшим выбором для нас будет Flask. Потому что он является простым, но мощным веб-фреймворком, который предназначен для быстрого и легкого начала работы с возможностью масштабирования до сложных приложений. Django, в свою очередь, является более тяжелым и монолитным веб-фреймворком из-за необходимости наличия всех встроенных приложений и компонент, с которыми он поставляется.

При сравнении этих двух фреймворков, Flask сравнивают со снорклингом, а Django – с дайвингом, что, на мой взгляд, является весьма наглядным сравнением (см. Рис 3.2).



Рис. 3.2 Flask vs Django by [@iconicbestiary](#)

Реализуем серверную часть нашего сервиса [\[1/web-server/server.py\]](#). Flask настолько прост в использовании, что реализация нашей задумки едва ли займет 10 строк. Установим данный фреймворк, импортируем из него необходимые модули и определим экземпляр основного приложения:

```
import cv2
import json
import numpy as np
from flask import Flask, request, Response
app = Flask(__name__)
```

Далее определим маршруты, по которым будет осуществляться доступ к логике веб-приложения, основным из которых будет “/face_detection”, обрабатывающий POST запросы в соответствии с требованиями HTTP RFC [\[9\]](#):

```
@app.route('/face_detection', methods=['POST'])
def detection():
    nparray = np.frombuffer(request.data, np.uint8)
    img = cv2.imdecode(nparray, cv2.IMREAD_COLOR)
    faces = ... # BeautyDeepNet detection
    return Response(
        response=json.dumps({'faces': faces}),
        status=200,
        mimetype='application/json'
    )
```

Запустим приложение, указав в имени прослушивающего хоста значение “0.0.0.0”, чтобы сервер был доступен извне, а также значение порта и опцию обработки запросов многопоточном режиме:

```
app.run(host='0.0.0.0', port=5000, threaded=True)
```

В результате чего сервер будет доступен со всех адресов IPv4 в локальной сети по публичному IP адресу машины, на которой он запущен (<http://192.168.0.device:5000/>).

При обращении к веб-приложению посредством метода POST мы получим необходимые данные о лицах на заданном изображении в виде JSON-объекта, который впоследствии можно будет легко обработать клиентом мобильного приложения.

3.3 Развертывание веб-приложения

Многие провайдеры облачного хостинга предлагают удалённую управляемую платформу, на которой могут работать любые приложения. Всё, что нужно предоставить для развертывания проекта на этих платформах, – это сам проект, потому что вычислительные мощности, операционные системы, интерпретаторы и компиляторы различных языков программирования, хранилища данных и так далее – всё это предоставляется и обслуживается самой облачной платформой.

Однако, исходя из опыта моего предыдущего проекта [10], развёртывание приложений на фриимиум-платформах заканчивается либо внезапной остановкой всех задействованных ресурсов проекта, либо принудительной активацией платной подписки по окончании пробного периода или при превышении определённых квот.

Дабы не тратить денежные ресурсы, активируя биллинг-аккаунт той или иной облачной платформы, и энергетические ресурсы, создавая новые аккаунты по истечению пробных периодов на старых, усовершенствуем наше веб-приложение для лёгкого и быстрого развёртывания на любом бесплатном сервисе облачных вычислений, например, Google Cloud Shell, Google Collab, AWS Cloud Service и так далее.

Самый простой способ демонстрации веб-приложений с любого компьютера, будь то виртуальная или локальная машина, – это использование сервиса для туннелирования локального хоста. В случае фреймворка Flask воспользуемся службой Ngrok [11], которая позволит нашему веб-приложению, работающему на любом удалённом хосте, быть доступным через Интернет из любой точки мира по временной ссылке.

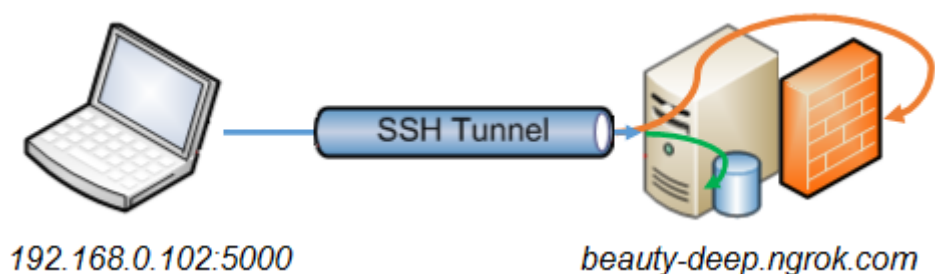


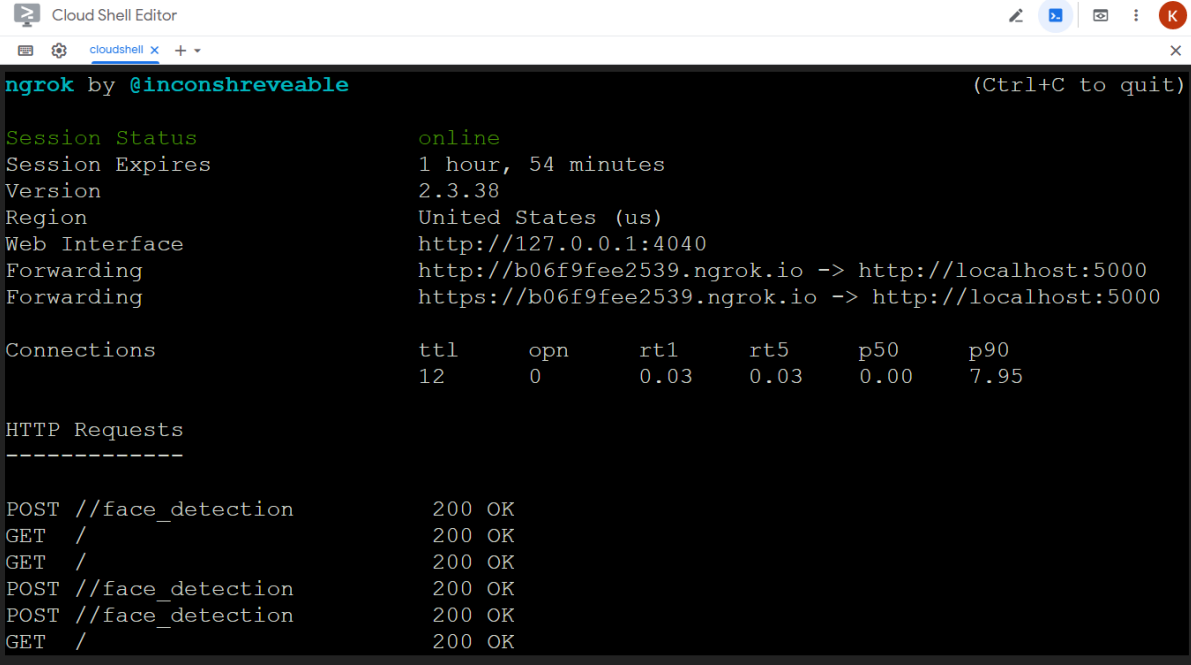
Рис 3.3 Схема туннелирования

Для этого установим службу flask-ngrok и добавим её в наше веб-приложение [\[1/web-server/ngrok.py\]](#):

```
from flask import Flask
from flask_ngrok import run_with_ngrok
app = Flask(__name__)
run_with_ngrok(app)
```

Создадим файл зависимостей нашего веб-приложения [\[1/web-server/requirements.txt\]](#) и скрипт [\[1/web-server/setup.sh\]](#), разворачивающий его на любом сервере на базе Linux.

Для проверки корректности работы проекта, развернем веб-приложение с помощью бесплатного Google Cloud Shell и протестируем некоторые HTTP-запросы (см. Рис. 3.4).



The screenshot shows the Google Cloud Shell Editor interface. At the top, it says 'Cloud Shell Editor' and 'cloudshell x +'. Below that, the title bar reads 'ngrok by @inconshreveable' and '(Ctrl+C to quit)'. The main content area displays the status of the ngrok service. It includes a 'Session Status' section with details like 'online', '1 hour, 54 minutes' remaining, version '2.3.38', region 'United States (us)', web interface 'http://127.0.0.1:4040', and forwarding rules. Below this is a 'Connections' table with columns for ttl, opn, rt1, rt5, p50, and p90. The table shows 12 connections with 0 open, and various response times. Finally, there is an 'HTTP Requests' section showing a list of requests and their responses, all of which are '200 OK'.

```
ngrok by @inconshreveable (Ctrl+C to quit)

Session Status      online
Session Expires    1 hour, 54 minutes
Version             2.3.38
Region              United States (us)
Web Interface       http://127.0.0.1:4040
Forwarding          http://b06f9fee2539.ngrok.io -> http://localhost:5000
Forwarding          https://b06f9fee2539.ngrok.io -> http://localhost:5000

Connections         ttl    opn    rt1    rt5    p50    p90
                   12     0     0.03   0.03   0.00   7.95

HTTP Requests
-----
POST //face_detection 200 OK
GET /                  200 OK
GET /                  200 OK
POST //face_detection 200 OK
POST //face_detection 200 OK
GET /                  200 OK
```

Рис. 3.4 Flask Ngrok на Google Cloud Shell

Веб-приложение полностью развернулось самостоятельно примерно за полторы минуты и успешно обработало большое количество запросов на обработку различных изображений через службу туннелирования Flask Ngrok, что является отличным результатом в рамках поставленной задачи.

В дополнение можно сказать, что благодаря фреймворку Flask, реализованное веб-приложение легко масштабировать: например, оформить главную страницу, добавив к ней различные компоненты и дизайн, или внедрить в функционал технологии BeautyDeepNet нейротипологический анализ лиц и различные маски, и так далее.

ГЛАВА 4. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ МОБИЛЬНОГО ПРИЛОЖЕНИЯ

4.1 Постановка задачи

В этой главе мы продолжим проектирование и реализацию сервисной инфраструктуры проекта BeautyDeer, затронув в частности его клиентскую часть, а именно — мобильное приложение под платформу Android. В ходе реализации приложения рассмотрим довольно известный, очень удобный, мощный и многофункциональный фреймворк для кроссплатформенной разработки с открытым исходным кодом, отличной документацией и самым отзывчивым русскоязычным и не только комьюнити – Kivy [\[12\]](#).

В дополнение к основному функционалу по оценке красоты человеческих лиц на фотографии добавим анализ некоторых параметров: формы подбородка, носа, бровей, глаз, губ, лица в целом, наличие улыбки, золотого сечения, общей симметрии и поворота лица. Для разнообразия и некоторой забавы попробуем угадать характер человека по его лицевым параметрам, дадим общую характеристику красоты его лица, а также рассчитаем процент уникальности его привлекательности, построив наглядное распределение соотношений красоты, исходя из статистических данных.

Спроектируем интуитивно понятный и визуально привлекательный интерфейс мобильного приложения, добавив к нему информационный раздел, мультиязычность, выбор цветовых тем, серверные настройки и другие дополнительные опции.

В заключение соберем, скомпилируем и упакуем итоговую реализацию мобильного интерфейса в арк-файл, содержащий весь код, ресурсы и нативные библиотеки приложения, с помощью Kivy Buildozer [\[13\]](#), в результате чего приложение можно будет установить на любой управляемый операционной системой Android смартфон.

4.2 Обработка и анализ лицевых параметров

Как уже было описано в предыдущей главе, как только веб-клиент обработает отправленную пользователем фотографию нейронной сетью, в качестве результата он вернёт все найденные лица, координаты их ключевых точек (глаз, бровей, носа, губ и подбородка) и процент привлекательности. Для удобной обработки и анализа полученных лицевых параметров создадим класс Face [\[1/mobile-app/face_params.py\]](#), который будет содержать всю информацию об этих параметрах, а также вычислять по ним необходимые метрики:

- 1) Форма глаз (круглые, миндалевидные, суженные) – определяется по соотношению ширины глаз и их открытости;
- 2) Форма бровей (горизонтальные, восходящие, дугообразные) – определяется по углу кривизны каждой из бровей;
- 3) Форма носа (длина: короткий, средний длинный; ширина: узкий, суженный, широкий) – определяется по соотношению ширин и длин носа и глаз соответственно;
- 4) Форма губ (тонкие, естественные, пухлые) – определяется по соотношению их ширины и длины;
- 5) Форма подбородка (острый, закруглённый, квадратный) – определяется по углу его остроты;
- 6) Форма лица (округлая, вытянутая) – определяется по отношению ширины лица к его длине;
- 7) Наличие улыбки (в процентном соотношении: чем больше процент, тем улыбка более выражена) – определяется по разнице между изгибами верхней и нижней губы;
- 8) Золотое сечение (в процентном соотношении: чем больше процент, тем ближе лицо к золотому сечению) – лицо является близким к идеальному или близким к золотому сечению, если отношения его длины к ширине, ширины рта к ширине носа и расстояния между зрачками к расстоянию между бровями равно числу Φ ;
- 9) Симметричность лица (в процентном соотношении: чем больше процент, тем выше симметричность лица) – лицо является симметричным, если расстояние между глазами, ширина носа и расстояние между бровями приблизительно равны (то есть можно провести две параллельные линии, касающиеся кончиков бровей, глаз и носа), а также расстояние от горизонтальной линии,

касающейся всех кончиков бровей, до кончика носа должно быть равно расстоянию от нижней части подбородка до кончика носа;

- 10) Поворот лица (в процентном соотношении: чем больше процент, тем больше поворот лица в сторону профиля) – определяется по соотношению расстояний от правого и левого висков до середины носа;
- 11) Характер личности (максимально нейтральные и присущие каждой личности примерно в равной степени прилагательные) – определяется уникально под каждую комбинацию вышеперечисленных параметров;
- 12) Характеристика привлекательности (специально подобранные прилагательные для характеристики красоты лица) – определяется в зависимости от процента привлекательности:
 - 0%: хуже некуда; 5%: отвратительное;
 - 10%: страшное; 15%: отталкивающее;
 - 20%: ужасное; 25%: некрасивое;
 - 30%: неприятное; 35%: заурядное;
 - 40%: невыразительное; 45%: обыкновенное;
 - 50%: симпатичное; 55%: милое;
 - 60%: интересное; 65%: обаятельное;
 - 70%: привлекательное; 75%: очаровательное;
 - 80%: красивое; 85%: восхитительное;
 - 90%: великолепное; 95%: ослепительное;
 - 100%: идеальное.
- 13) Уникальность внешности (приблизительный процент людей со схожей оценкой красоты) – вычисляется исходя из нормального распределения привлекательности людей со средним значением по красоте ≈ 56 и стандартным отклонением ≈ 14 ;

Таким образом, благодаря данному классу, мы будем получать не только процент привлекательности лиц и их ключевые точки, но также вычислять вышеперечисленные метрики для анализа лицевых параметров, чтобы потом представить их в удобном для пользователя формате на соответствующем экране мобильного приложения.

Создадим ещё один вспомогательный класс `Image` [\[1/mobile-app/image_processing.py\]](#), который возьмёт на себя коммуникацию с веб-приложением, пред- и постобработку входного изображения, а также обработку и анализ лицевых параметров при помощи уже реализованного класса `Face` [\[1/mobile-app/face_params.py\]](#). Экземпляр класса `Image` будет принимать путь к изображению в файловой системе и иметь следующие методы:

- 1) `load_image` – для загрузки и предобработки входного изображения;
- 2) `send_request` – для коммуникации с веб-приложением по обработке входного изображения нейронной сетью и формирования списка лиц (экземпляров класса `Face`) для дальнейшей обработки и анализа их параметров;
- 3) `create_output` – для создания изображений-результатов с возможностью наложить маску ключевых точек лица и выделить индекс лица, для его последующего выбора и просмотра вычисленных для него метрик.

Таким образом, получить, к примеру, описание параметра симметричности лица с индексом 2 (функционал выбора конкретного лица на изображении реализуется уже в описании логики экрана `BeautyScreen` мобильного приложения) на необходимом языке и изображение-результат с масками ключевых точек можно, передав классу `Image` путь к входной фотографии:

```
face_id = 2
im = Image(image_path)
im.send_request(server_url)
im.create_output(mask=True)
im.faces[face_id].get_symmetry_param(language_id)
```

Следует отметить, что все языковые настройки будущего интерфейса мобильного приложения будут храниться в специальном языковом словаре [\[1/mobile-app/app_languages.py\]](#), с помощью которого, обратившись по ключу необходимого элемента интерфейса, можно получить его копию, переведённую на один из поддерживаемых словарём языков (Русский, Английский, Немецкий).

4.3 Проектирование основного интерфейса приложения

Для начала стоит сказать несколько слов об инструменте, которым мы будем пользоваться на протяжении проектирования интерфейса и реализации его основных логик для нашего проекта.

Kivy – это кроссплатформенный фреймворк с открытым исходным кодом для разработки графических интерфейсов на Python. Благодаря Python при работе с Kivy нам гарантирована скорость и комфортность разработки, лаконичность кода и возможность моментально вносить изменения и тестировать их. Благодаря кроссплатформенности наше единожды написанное приложение может легко запуститься и работать на всех доступных платформах: Linux, Windows, OS X, Android, iOS и Raspberry Pi. Согласно Google, 99.9% устройств соответствуют требованиям для запуска приложений на Kivy. Также Kivy является зрелым, но быстроразвивающимся фреймворком, с отлично написанной документацией, содержащей наглядные примеры и визуализации, а также большим, активным и дружелюбным комьюнити, с которым лично я немало пообщался в ходе разработки приложения.

Само мобильное приложение назовем соответственно BeautyDeer, и для его реализации будем использовать Kivy [\[12\]](#) версии 1.11.1 с набором виджетов в стиле Material Design [\[14\]](#) библиотеки KivyMD [\[15\]](#) и её расширением AKivyMD [\[16\]](#). Стоит отметить, что на начальном этапе реализации приложения наиболее стабильной версией фреймворка являлась версия 1.11.1, но на данный момент выпущена более новая стабильная версия 2.0.0 с многими исправлениями и изменениями. Полная реализация мобильного приложения со всеми подробностями описываться не будет, так как это очень масштабная работа, однако всё необходимое можно найти и изучить в одноимённом каталоге моего репозитория [\[1/mobile-app\]](#) с хорошо задокументированным и понятным кодом.

Сам фреймворк Kivy предоставляет свой собственный язык разметки, специально ориентированный на простоту и масштабируемость проектирования дизайна графического интерфейса. Этот язык позволяет легко отделить дизайн интерфейса, описываемого в соответствующих .kv файлах, от логики приложения, описываемой в соответствующих .py файлах.

В основном, проектирование дизайна состоит в описании необходимых виджетов и макетов, а реализация логики – в описании событий и свойств к ним. Для простоты понимания язык разметки Kivy похож на YAML по синтаксису и на CSS/HTML по функционалу, однако Kivy понимается и ощущается в разы проще.

Создадим макет основного интерфейса на базе ScreenManager, который будет содержать все экраны нашего приложения и задавать их основные свойства:

```
class MainInterface(ScreenManager):
    '''Contains all screens and their settings as a ScreenManager'''
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.toolbar.md_bg_color = 1, 1, 1, 1
        self.toolbar.specific_text_color = 1, 1, 1, 1
        self.bottom_panel.panel_color = .27, .84, 1, 1
    def set_toolbar_color(self, color):
        self.toolbar.md_bg_color = color
    def set_toolbar_text_color(self, color):
        self.toolbar.specific_text_color = color
    def set_bottom_panel_color(self, color):
        self.bottom_panel.panel_color = color
```

Соответствующая разметка будет содержать экземпляры основных виджетов и экранов нашего приложения и выглядеть следующим образом:

```
<MainInterface>:
    toolbar: toolbar
    bottom_panel: bottom_panel
    home_screen: home_screen
    image_screen: image_screen
    settings_screen: settings_screen
    spinner_screen: spinner_screen
    beauty_screen: beauty_screen
```

Как уже видно по файлу разметки, основной интерфейс будет содержать следующие компоненты (см. Рис. 4.1):

- 1) Тулбар с названием и логотипом приложения вверху интерфейса;
- 2) Навигационная панель с тремя кнопками, перенаправляющими на соответствующие экраны внизу интерфейса
- 3) Один из экранов, в зависимости от выбора пользователя, в центре интерфейса:

- Домашний экран, содержащий рекомендательную, справочную и общую информацию о приложении;
- Экран для загрузки изображения из файловой системы или камеры смартфона;
- Экран настроек приложения, содержащий различные опции по выбору темы, языка, сервера и так далее;
- Экран ожидания, который будет всплывать на этапе загрузки и обработки изображения;
- Экран результатов, на котором будет изображена соответствующая информация о привлекательности найденных лиц, их параметры и различные иллюстрации с комментариями.

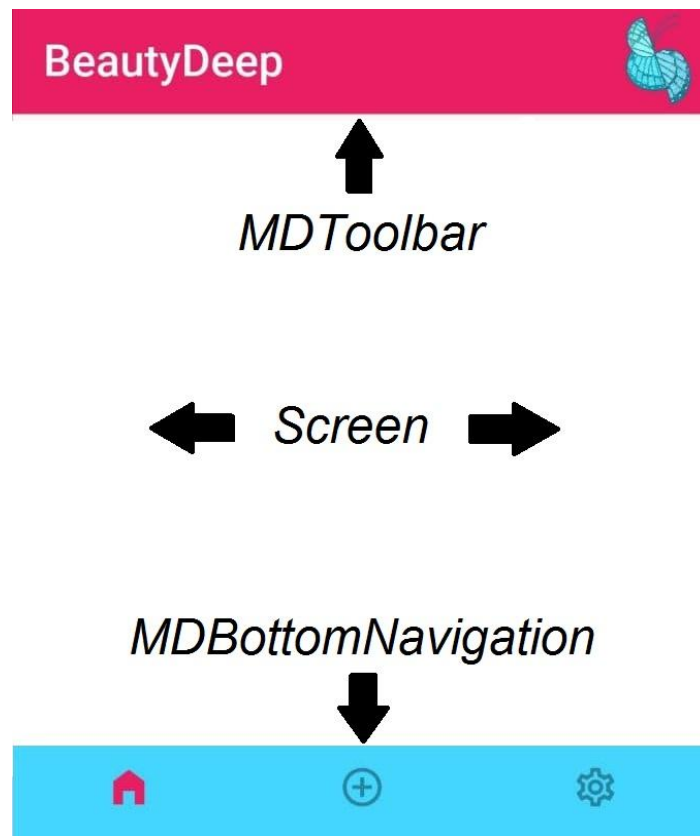


Рис. 4.1 Компоненты основного интерфейса

Соответствующие разметки для тулбара и навигационной панели имеют вид:

```
MDToolbar:
    id: toolbar
    title: 'BeautyDeep'
    right_action_items: [['images/logo.png', lambda x: None]]

MDBottomNavigation:
    id: bottom_panel
    HomeScreen:
        id: home_screen
        name: 'home_screen'
        icon: 'home-variant'
        ...
    ImageScreen:
        id: image_screen
        name: 'image_screen'
        icon: 'plus-circle-outline'
        on_tab_release: app.file_manager_open()
        ...
    SettingsScreen:
        id: settings_screen
        name: 'settings_screen'
        icon: 'cog-outline'
        ...
```

У каждого описываемого нами виджета, которых в Kivy и KivyMD огромное многообразие, есть свои собственные уникальные настраиваемые параметры типа позиции и ориентации виджета относительно других компонент на экране, цвета, стиля шрифтов, величины отступов, откликов на различные события, иконок и прочее. Все эти параметры легко настраиваются и описываются с помощью языка разметки. Обратившись по идентификатору или имени отдельного виджета, описанного в .kv файле, мы можем легко добавить к нему различные события и свойства, создав одноимённый класс, реализующий необходимые для поставленной задачи методы. Таким образом Kivy даёт нам полную свободу как в проектировании дизайна, так и в реализации бизнес логики приложения.

В качестве примера рассмотрим проектирование и реализацию домашнего экрана мобильного приложения, на котором будет располагаться вся основная и справочная информация проекта.

Создадим класс на базе класса навигационной панели, содержащий логику нашего экрана по отображению раскрывающихся параграфов с необходимой информацией. Основными полями класса будут заголовки параграфов и их контент на соответствующем языке, а также иконки, иллюстрирующие основную суть передаваемой в параграфах информации:

```
class HomeScreen(MDBottomNavigationItem):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.titles = None
        self.texts = None
        self.icons = [
            'rocket-home.png', 'phone-home.png', 'cog-home.png',
            'beauty-home.png', 'face-home.png', 'photo-home.png'
        ]
```

При создании экземпляра класса домашнего экрана приложения, будет вызываться метод, добавляющий весь необходимый контент в виде виджетов на данный экран:

```
def add_content(self, lang):
    self.home_content.clear_widgets()
    self.titles = languages[lang]['home_titles']
    self.texts = languages[lang]['home_texts']
    for tls, txs, ics in zip(self.titles, self.texts, self.icons):
        content = MDBoxLayout(
            adaptive_height=True,
            orientation='vertical',
            padding=[dp(25), dp(100)]
        )
        content.add_widget(
            MDLabel(
                text=txs,
                size_hint=(1, None),
                height=dp(Window.height * .1),
                theme_text_color='Secondary'
            )
        )
    self.home_content.add_widget(MDExpansionPanel(
        icon='images/' + ics,
        content=content,
        panel_cls=MDExpansionPanelOneLine(text=tl)
    ))
```

Заключительным шагом в создании домашнего экрана, добавим в соответствующее описание HomeScreen .kv файла область прокрутки и макет одноколоночной сетки, на которой будут располагаться информационные разделы в виде виджетов.

HomeScreen:

```
...
ScrollView:
    MDGridLayout:
        id: home_content
        cols: 1
        adaptive_height: True
```

После чего спроектированный экран будет выглядеть следующим образом (см. Рис. 4.2):

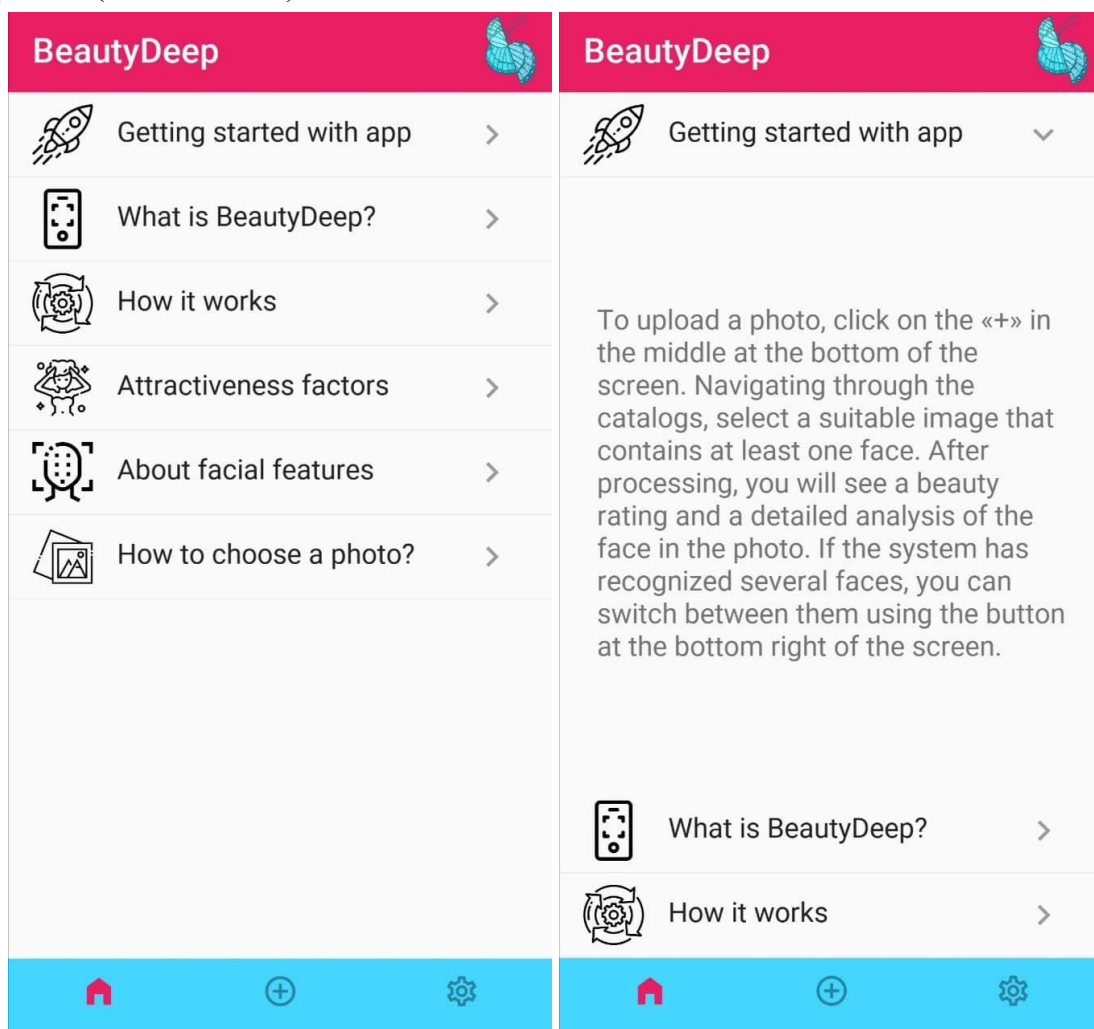


Рис. 4.2 Домашний экран мобильного приложения

Примерно аналогичным образом проектируются и остальные экраны, составляющие наше мобильное приложение (см. Рис. 4.3).

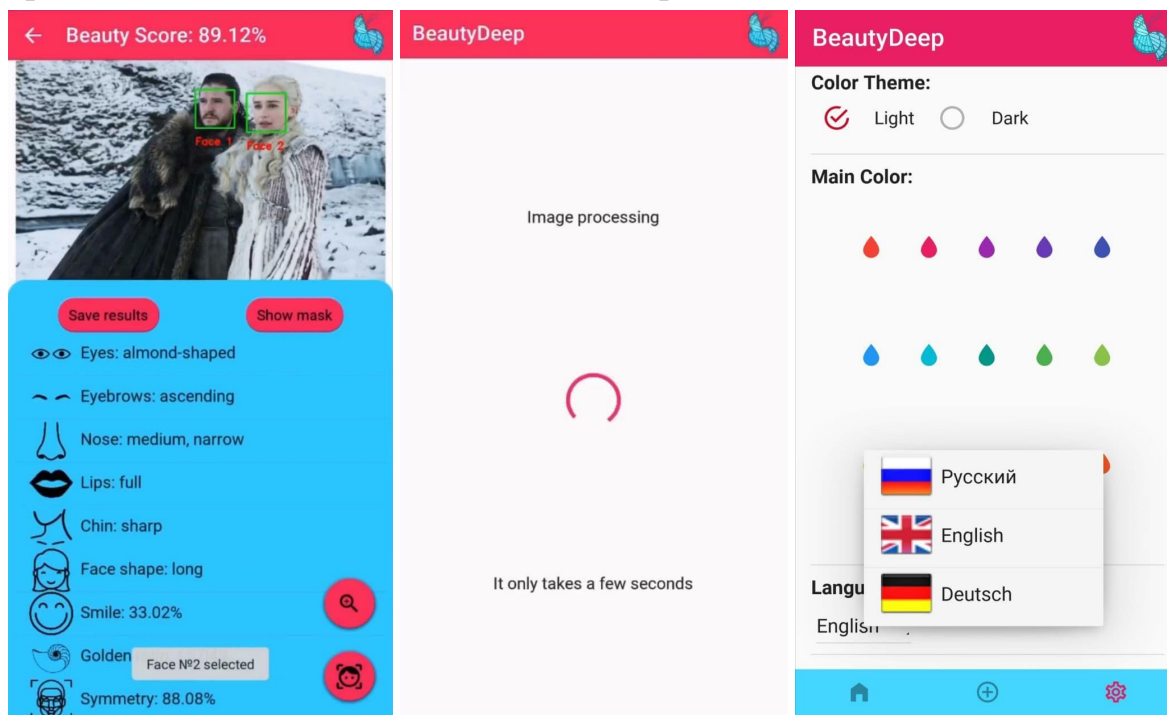


Рис. 4.3 Экраны результатов, загрузки и настроек приложения

4.4 Реализация основной логики приложения

Вся основная логика мобильного приложения будет содержаться в классе `BeautyDeerApp`, наследующем все необходимые технические и графические составляющие от базового класса Kivy Material Design – `MdApp`. Этот класс является базовым для всех приложений KivyMD, и благодаря ему начинается и поддерживается основной жизненный цикл любого проекта на любой платформе.

В свою очередь, сам класс `BeautyDeerApp` будет хранить информацию обо всем мобильном приложении и его интерфейсе в целом, включая стиль и цвет темы, язык интерфейса, URL или IP адрес сервера, который будет обрабатывать изображения, основные пути к директориям проекта и прочее. Все пользовательские настройки приложение будет хранить в конфигурационном файле формата JSON, находящемся во внутренней памяти девайса.

Инициализатор класса `BeautyDeerApp` будет иметь следующий вид:

```

class BeautyDeepApp(MDApp):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.store = JsonStore('config.json')
        if not self.store:
            self.store['user_config'] = {
                'theme_style': 'Light',
                'theme_color': 'Pink',
                'app_language': 1,
                'public_ip': 'http://192.168.0.102:5000'
            }
        self.language = self.store['user_config']['app_language']
        self.server_ip = self.store['user_config']['public_ip']
        self.APP_ROOT = os.path.abspath('')
        self.DCIM = self.get_dcim_path()

```

Также основным класс мобильного приложения будет реализовывать несколько важных методов:

- 1) build – подготавливает конфигурации приложения, добавляет весь контент на экраны и инициализирует приложение в виде дерева виджетов;
- 2) on_start – запрашивает разрешения системы на доступ в интернет, чтение и запись во внутреннее хранилища устройства;
- 3) on_pause – при переключении на другое приложение, наше будет работать в фоновом режиме и ждать пока пользователь не переключится обратно;
- 4) on_resume – при переключении с другого приложения на наше, оно возобновит работу из предыдущего состояния;
- 5) on_stop – сработает при любом завершении приложения и удалит его кэш и временные файлы;
- 6) get_dcim_path – определит каталог галереи с изображениями по умолчанию на любой платформе;
- 7) switch_theme_style – сменит тему мобильного приложения на выбранную пользователем;
- 8) switch_theme_color – сменит цветовую палитру мобильного приложения на выбранную пользователем;
- 9) change_language – сменит язык интерфейса на выбранный пользователем;
- 10) change_server_ip – установит настройки IP сервера для обработки изображений нейронной сетью;

- 11) `file_manager_open` – откроет файловый менеджер по умолчанию для выбора изображения из галереи;
- 12) `select_path` – передаст выбранное изображение обработчику и переключит менеджер экранов на экран ожидания;
- 13) `set_current_screen` – переключит менеджер экранов на указанный экран;
- 14) `get_faces` – перешлёт изображение на удалённый сервер для его обработки, вычислит необходимые метрики по полученным результатам и отобразит их на соответствующем экране;

Для того, чтобы запустить и протестировать реализованное приложение в графическом интерфейсе Kivy, необходимо импортировать все зависимости и библиотеки проекта и вызвать метод `run()` базового класса `BeautyDeepApp`:

```
if __name__ == '__main__':  
    BeautyDeepApp().run()
```

Видеопревью с демонстрацией функционала разработанного мобильного приложения можно посмотреть на главной странице GitHub-репозитория [\[1\]](#) или в каталоге с документацией проекта [\[1/papers/app-preview.mov\]](#).

4.5 Сборка приложения под платформу Android

Чтобы запустить реализованное нами приложение `BeautyDeep` на любом смартфоне, под управлением операционной системой Android, нам необходимо подготовить для него установочный пакет. Самым доступным и популярным решением этой задачи будет использование `Kivy Buildozer`.

`Buildozer` – это инструмент, предназначенный для простой сборки мобильных приложений под различные платформы. С его помощью можно автоматизировать весь процесс вплоть до загрузки, настройки, построения и тестирования необходимых компонент, таких как `python-for-android`, `Android SDK`, `NDK` и других. После чего упакованное в `apk`-файл приложение можно автоматически отправить на устройство для его дальнейшей отладки.

Также Buildozer имеет разделенный на этапы процесс построения и упаковки приложения, каждый из которых кэшируется и может быть перестроен при внесении каких-либо изменений.

С помощью следующих команд можно установить Buildozer в операционную систему Linux:

```
git clone https://github.com/kivy/buildozer.git
cd buildozer
sudo python3 setup.py install
```

Как только Buildozer будет установлен, необходимо проинициализировать его в корневом каталоге нашего проекта с мобильным приложением:

```
git clone https://github.com/Defaultin/BeautyDeep.git
cd BeautyDeep/mobile-app/
buildozer init
```

После инициализации в том же каталоге с приложением создастся файл `buildozer.spec`, который будет содержать все необходимые конфигурации и управлять сборкой нашего проекта. Мы можем отредактировать его необходимым образом, установив нужные нам настройки, влияющие на процесс сборки и получившееся приложение в целом. Рассмотрим некоторые важные для сборки нашего проекта конфигурации файла `buildozer.spec` [\[1/mobile-app/buildozer.spec\]](#) и установим для них соответствующие значения:

- 1) Название приложения (*title = Beauty Deep*);
- 2) Каталог с исходным кодом мобильного приложения, в котором находится файл `main.py` (*source.dir = .*);
- 3) Расширения исходных файлов, используемых для сборки (*source.include_exts = py,png,jpg,jpeg,ttf,kv*);
- 4) Список каталогов встраиваемых в приложение шаблонов и ресурсов (*source.include_patterns = images/**);
- 5) Последняя версия сборки приложения (*version = 3.0.0*);
- 6) Список всех зависимостей приложения (*requirements = python3==3.7.6, hostpython3==3.7.6, kivy==1.11.1, plyer, kivymd, akivymd, sdl2_ttf==2.0.15, numpy, requests, urllib3, chardet, certifi, idna, simplejson, opencv*);
- 7) Преплеш-файл или загрузочная заставка приложения (*presplash.filename = %(source.dir)s/images/logo-bg.png*);

- 8) Файл мини-иконки приложения (*icon.filename = %(source.dir)s/images/logo.png*);
- 9) Ориентация интерфейса приложения (*orientation = portrait*);
- 10) Цвет фона загрузочной заставки приложения (*android.presplash_color = #E91E63*);
- 11) Разрешения доступа к операционной системе Android (*android.permissions = INTERNET, READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE*);
- 12) Каталог, содержащий необходимый Android SDK для сборки (*android.sdk_path = ../android-sdk/*);
- 13) Уровень логирования ошибок при отладке (*log_level = 2*).

Далее, подключив смартфон в режиме разработчика к Linux машине с уже установленным и сконфигурированным инструментом Buildozer, запустим автоматическую сборку с последующей отладкой мобильного приложения утилитой logcat (*adb logcat *:S python:D*) на нашем телефоне с помощью следующей команды:

```
buildozer android debug deploy run logcat
```

Если сборка и отладка приложения прошла успешно, то можно собрать финальный арк-файл, который уже можно устанавливать на любой девайс под управлением операционной системы Android, а также загружать в PlayMarket или в другой магазин мобильных приложений:

```
buildozer android release
```

Готовое приложение версии 3.0.0 можно найти в основном каталоге репозитория проекта BeautyDeer [\[1\]](#), установить себе на смартфон, протестировать и поиграться.

Вес установочного пакета получился всего 34.8 MB, но его можно уменьшить примерно в два раза, путём удаления лишних зависимостей Python-интерпретатора и неиспользуемых модулей Kivy, а также сжатия некоторых медиа файлов.

Стоит отметить, что благодаря кроссплатформенности Kivy и возможностям Buildozer, наше приложение можно легко собрать под любую поддерживаемую платформу, например iOS, и без проблем запустить его.

ЗАКЛЮЧЕНИЕ

В данной работе был представлен процесс проектирования и разработки сервисной архитектуры BeautyDeer по оценке и анализу различных параметров человеческих лиц, изображённых на фотографии, на основе:

- нейросетевой модели, обученной на нескольких тысячах фотографий реальных людей различной внешности и национальности;
- мобильного приложения под управлением операционной системы Android, выступающего в качестве удобного и понятного пользовательского интерфейса;
- веб-клиента, связывающего интерфейс приложения с основной логикой обработки и анализа изображений.

Проект BeautyDeer актуален и жизнеспособен как самостоятельный продукт, однако все реализованные в нём технологии могут также использоваться в других системах, как коммерческих, так и научных: например, в качестве дополнительной функции для соцсетей и онлайн платформ, или как инструмент для анализа данных в социологии.

Вся проделанная работа по своему масштабу соизмерима с крупным проектом целой команды разработчиков: он требует не только определённых технологий и ресурсов, но и продвинутых навыков в машинном обучении, построении бизнес-процессов для сложных архитектур, разработке мобильных и веб приложений, cloud-технологиях, анализе данных и других областях сферы информационных технологий.

Проект BeautyDeer можно развивать и дальше – дорабатывать функционал (например, сделать более точным анализ характера и личностных качеств при помощи правил нейротипологии и физиогномики, или «научить» систему давать советы по увеличению уровня красоты, или даже осуществлять коррекцию лица в сторону улучшения его привлекательности), создавать маски для социальных сетей, адаптировать приложение под другие платформы и выводить его на рынок как самостоятельный продукт.

Все материалы к проекту BeautyDeer v3.0.0 доступны в GitHub-репозитории [\[1\]](#) под свободной лицензией GNU GPL v3.0.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

- [1] BeautyDeep <https://github.com/Defaultin/BeautyDeep/>
- [2] A Diverse Benchmark Dataset for Multi-Paradigm Facial Beauty Prediction
<https://github.com/Defaultin/BeautyDeep/blob/master/papers/SCUT-FBP5500%20A%20Diverse%20Benchmark%20Dataset%20for%20Multi-Paradigm%20Facial%20Beauty%20Prediction.pdf>
- [3] OpenCV <https://docs.opencv.org/master/>
- [4] Label Distribution Based Facial Attractiveness Computation by Deep Residual Learning
<https://github.com/Defaultin/BeautyDeep/blob/master/papers/Label%20Distribution%20Based%20Facial%20Attractiveness%20Computation%20by%20Deep%20Residual%20Learning.pdf>
- [5] Keras <https://keras.io/guides/>
- [6] Dlib <http://dlib.net/python/index.html>
- [7] Flask <https://flask.palletsprojects.com/en/1.1.x/>
- [8] Django <https://docs.djangoproject.com/en/3.2/>
- [9] HTTP RFC <https://www.ietf.org/rfc/rfc2068.txt>
- [10] FourierTransformBot
<https://github.com/Defaultin/FourierTransformBot>
- [11] Flask Ngrok <https://github.com/gstaff/flask-ngrok>
- [12] Kivy <https://kivy.org/> <https://kivy.org/doc/stable/>
- [13] Kivy Buildozer
<https://kivy.org/doc/stable/guide/packaging-android.html>
- [14] Material Design <https://material.io/design/introduction>
- [15] KivyMD <https://kivymd.readthedocs.io/en/latest/>
- [16] AKivyMD <https://github.com/quitegreensky/akivymd>

АННОТАЦИЯ К РЕПОЗИТОРИЮ ПРОЕКТА

BeautyDeep:

- ❑ LICENSE
- ❑ README.md
- ❑ BeautyDeep-v3.0.0-pre-release.apk
- ❑ mobile-app
 - ❑ app_languages.py
 - ❑ beauty_deep.kv
 - ❑ buildozer.spec
 - ❑ face_params.py
 - ❑ image_processing.py
 - ❑ main.py
 - ❑ requirements.txt
 - ❑ images
- ❑ neural-network
 - ❑ data_extraction.py
 - ❑ generate_data.py
 - ❑ generate_data_norecompute.py
 - ❑ requirements.txt
 - ❑ test.py
 - ❑ train.py
- ❑ papers
 - ❑ app-preview.gif
 - ❑ app-preview.mov
 - ❑ BeautyDeep.pdf
 - ❑ Label Distribution Facial Attractiveness Computation.pdf
 - ❑ pipeline.png
 - ❑ Dataset for Multi-Paradigm Facial Beauty Prediction.pdf
- ❑ web-server
 - ❑ beauty_deep_net.py
 - ❑ ngrok.py
 - ❑ requirements.txt
 - ❑ server.py
 - ❑ setup.sh