

Instagram Data Analysis

PROJECT WORK ON INTELLIGENT SYSTEMS M

ALBERTAZZI RICCARDO - 0000796456

2017

Objectives

- Analysis of a Instagram dataset containing images and meta-data (user info, likes, comments, hashtags, ...)

In particular, the main task of this project has been

- Predict the number of likes of a given Instagram post using deep-learning techniques

Software...

- Ubuntu 17.04
- Python: de-facto language for machine learning and data analysis. Version 3.6
- Keras 2.0
 - High-level Python library for Deep Learning
 - Supports several backend frameworks, such as Tensorflow and Theano
 - Supports GPU computing (CUDA)
 - Build a Deep Learning model in a few minutes 😊
 - Lots of hyper-parameters 😞

... and hardware

- I5-2500k 4C 3.3Ghz (Sandy Bridge)
- 8 GB RAM
- NVIDIA GTX 580
 - Good video card in 2011 but not now: 1.5 GB VRAM, does not support Cuda 3.0 (required by Tensorflow and cuDNN) ☹
 - Theano supports Cuda 2.0 ☺ -> default backend for hardware acceleration in my project



FIRST STEPS

PYTHON, KERAS AND FAMOUS DATASETS

A solid orange horizontal bar at the bottom of the slide.

First steps

- Learning Python
- Learning NumPy
- Learning Keras
- Develop simple models for well known datasets
 - Mix of online tutorials and personal trial/error for fine tuning of the hyper-parameters

Keras

- Sequential API allow to build a deep learning model in a bunch of lines of code
- Each layer is abstracted by a Keras object
- Most useful layers:
 - **Dense**: fully connected layer. We can specify the number of neurons
 - **Dropout**: applies dropout to the input layer. We can specify the percentage of input connections to be dropped
 - **Flatten**: flattens the input layer. Useful as a bridge between convolutional and dense layers
 - **Conv2D**: 2D convolutional layer. We can specify the number of filters, the kernel size, the stride and the padding
 - **MaxPooling2D**: performs max pooling operation. We can specify the pool size and the stride
- Dense and Conv2D layers allow to specify the **activation function** (linear, relu, sigmoid, softmax)

Keras

- When compiling the model, we can specify the loss function to minimize during training.
- Important loss functions:
 - Mean squared error (mse)
 - Mean absolute error (mae)
 - Mean absolute percentage error (mape)
 - Binary crossentropy: used in binary classification
 - Categorical crossentropy: used in classification with more than 2 classes
- When training the model, we can specify:
 - Epochs: the number of iterations over the whole dataset
 - Batch size: number of samples per gradient update. Batch size can influence the final accuracy and the training time
 - Validation data: test data that will only be evaluated. Provides useful output to handle overfitting

Auto MPG Dataset

- The data concerns city-cycle fuel consumption in miles per gallon, to be predicted in terms of 8 car attributes (cylinders, displacement, horsepower, weight, acceleration, model year, origin, car name)
- Built in 1983: poor prediction capability for today car consumption!
- Obtained 4% mean absolute error with these 6 lines of code:

```
# Build the model
model = Sequential()
model.add(Dense(40, input_dim=7, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile model
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mae', 'mape'])

# Fit the model
model.fit(X_train, y_train, epochs=1500, batch_size=30, validation_data=(X_test, y_test))
```

Pima Indians Diabetes Dataset

- Donated in 1990 from National Institute of Diabetes and Digestive and Kidney Diseases
- Predict the onset of diabetes based on 8 diagnostic measures
- Again, reached 75%/80% accuracy with a few lines of code:

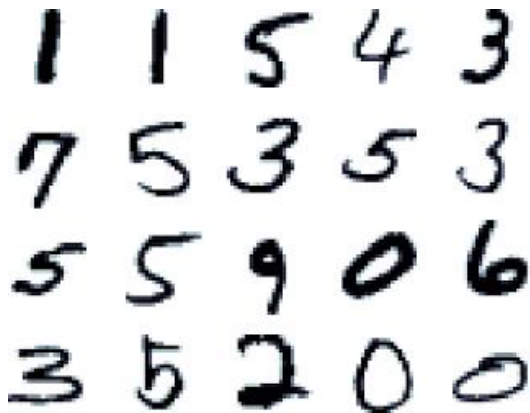
```
model = Sequential()
model.add(Dense(8, input_dim=8, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

# Fit the model
model.fit(X_train, y_train, epochs=1500, batch_size=30, validation_data=(X_test, y_test))
```

MNIST Dataset

- Database of image 28x28 pixels containing a handwritten digit [0-9]
- First approach with Convolutional Neural Networks
- 78% accuracy with a low number of layers



```
# 7. Define model architecture
model = Sequential()

model.add(Convolution2D(32, 3, 3, activation='relu', input_shape=(1,28,28)))
model.add(Convolution2D(32, 3, 3, activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

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

# 9. Fit model on training data
model.fit(X_train, Y_train,
          batch_size=32, nb_epoch=10, verbose=1)
```

Instagram like prediction

Instagram Dataset

- Found on a github project that had the same task but used traditional ML algorithms:
<https://github.com/gvsi/instagram-like-predictor>
- 16.5k rows containing:
 - User info (replicated in each row)
 - Username
 - Alias
 - Website
 - Url to Instagram profile
 - Number of Following
 - Number of Followers
 - Number of posts
 - Url to profile image
 - Image info
 - Publish date (all images belong to start 2017)
 - Url to the image
 - Mentions (other instagram users tagged)
 - Localization (only a few images have it)
 - List of hashtags
 - Description (caption below the image)
 - Number of likes

Around 800 different users

Dataset pre-processing

- Remove posts with multiple image
- Consider only users with less than 1 million followers (celebrities have been excluded)
- Remove a few images with 0 number of likes (problems during training)

First approach

- Predict likes based on meta-data only (not images)
- Information used:
 - Number of posts
 - Number of followers/following
 - Publish date and day of the week
 - Number of mentions
 - Number of hashtags
 - Hashtag analysis

First approach: a note on hashtag analysis

- How to measure hashtags?
- My simple approach:
 - Created a Python script to download the top 10k hashtags
 - Each hashtag has a value from 10k (most important) to 1
 - For each image I computed the sum of the used hashtags: each hashtag has weight [1,10k] if it's one of the top 10k hashtags, 0 if not
- Future work could analyze correlation between hashtags

First approach: results

- A very simple model, with dense layers only
- Loss function is mean absolute percentage error

```
model = Sequential()  
model.add(Dense(80, input_dim=COLUMNS-1, activation='relu'))  
model.add(Dense(30, activation='relu'))  
model.add(Dense(10, activation='relu'))  
model.add(Dense(1))
```

- Results
 - On training data: 54% (still too high)
 - On test data: orders of magnitude above! Overfitting and poor generalization capability

Second approach: CNN

- Use both meta-data and images
- Now each data contains two different inputs for the network
 - Meta-data input that requires dense/dropout layers
 - Image input that requires convolutional/pooling layers before dense layers
- Luckily Keras allows to define multi-input/multi-output networks using the Functional API
 - Similar to Sequential API, but we work at layer level: first we create the layer and connect them, then we create the model

Second approach: model building

#Convolutional neural network for images

```
image_input = Input(shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS), name='image_input')
image_nn = Conv2D(4, (5,5), activation='relu')(image_input)
image_nn = Conv2D(16, (3,3), activation='relu')(image_nn)
image_nn = MaxPooling2D(pool_size=(2,2))(image_nn)
image_nn = Conv2D(32, (3,3), activation='relu')(image_nn)
image_nn = Conv2D(64, (3,3), activation='relu')(image_nn)
image_nn = MaxPooling2D(pool_size=(2,2))(image_nn)
image_nn = Conv2D(128, (3,3), activation='relu')(image_nn)
image_nn = MaxPooling2D(pool_size=(2,2))(image_nn)
image_nn = Dropout(0.2)(image_nn)
image_nn = Conv2D(256, (3,3), activation='relu')(image_nn)
image_nn = MaxPooling2D(pool_size=(2,2))(image_nn)
image_nn = Dropout(0.2)(image_nn)
image_nn = Flatten()(image_nn)
```

#Neural network for user data and image metadata

```
data_input = Input(shape=(COLUMNS-1,), name='data_input')
data_nn = Dense(10, activation='relu')(data_input)
```

#Merge of the networks and last part of the network

```
output_nn = keras.layers.concatenate([image_nn, data_nn])
output_nn = Dense(500, activation='relu')(output_nn)
output_nn = Dropout(0.1)(output_nn)
output_nn = Dense(10, activation='relu')(output_nn)
output_nn = Dense(1, name='output')(output_nn)
```

#Model definition

```
model = Model(inputs=[image_input, data_input], outputs = [output_nn])
```

- This model still fits into the GPU!
(if a little batch size is used)


Second approach

1. Pre-processing: Python script to download the images and resize them to a fixed size (256x256)
2. Obviously it's not possible to load all images in memory before the training
 - Batch training
 - Keras allows to use a Python generator instead of raw input data
 - The generator loops on the whole training set and yields each time a batch of images and their corresponding meta-data
 - Keras performs training and loading of the next batch in parallel
3. Results: mean absolute percentage error on of 72% for both training and test set. We are going into the right direction 😊

Mean number of likes

- After these two approaches I considered a new measure: the **mean number of likes per user**
- One could say that I use the number of likes to predict the number of likes
- But this attribute is the most useful attribute to measure the number of likes, and the objective of this project is to predict the number of likes of a future image before posting it, so why not use it?
- If the prediction of our neural network was the mean number of likes, the loss function would immediately drop to 43%
- So any result with mape above 43% has to be considered a bad result

First and second approach revisited

- I added the attribute «mean number of likes per user» to the data available in the first approach (meta-data only) and trained again
 - The result is mape = 33% in training data and 45% in test data: the network still can't generalize enough! (Overfitting)
 - In the second approach (meta-data and images) the network performance is slightly better, but the result on the test data is 43%
- 
- The output is too dependent to the attribute 'mean number of likes'. This prevents the network from learning

Google Vision API

- The CNN itself is not very helpful. The mean number of likes has a strong dependency on the output number of likes and hides other useful information found by the network
- We need another way to get information about the image content
- Google has powerful APIs (Google Vision API) to analyze the content of an image
 - REST API and support for many languages (including Python 😊)
 - Very low cost: first 1000 units/month are free, then \$1.50 per 1000 units. But you have a free \$300 credit when you first create a Google Cloud Platform account
 - Google Vision API perform label detection, text detection, face detection, web search detection, logo detection, ...
 - I used **label detection** for this project. In particular I created a Python script that uses Google Vision API to upload the photos of my dataset and download the corresponding labels with their score

Google Vision API Example



Profession	86%
Speech	85%
Public Speaking	81%
Official	81%
Businessperson	72%
Spokesperson	66%
Suit	63%
Entrepreneur	57%
Professional	57%



Skyscraper	97%
Tower Block	95%
Metropolitan Area	95%
Building	95%
Condominium	94%
Metropolis	92%
Commercial Building	91%
Mixed Use	86%
Tower	86%
Corporate Headquarters	84%
Daytime	79%
Cityscape	73%

Autoencoder for labels

- Labels detected by Google Vision API are too specific: we need to generalize
 - We need a network that 'compresses' the labels information into a small number of neurons
- With Keras Functional API it's possible to build autoencoders too!
- An autoencoder is simply a model where the desired output is equal to the input

```
def build_model(self, encoding_dim = 50):
    print('Building labels encoder and autoencoder...')
    input_layer = Input(shape=(self.row_size,))
    encoded = Dense(encoding_dim, activation='relu')(input_layer)
    decoded = Dense(self.row_size, activation='sigmoid')(encoded)

    self.autoencoder = Model(input_layer, decoded)
    self.encoder = Model(input_layer, encoded)

    self.autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
    self.encoding_dim = encoding_dim

def train(self, epoch=50, batch=32):
    print('Training labels autoencoder...')
    self.autoencoder.fit(self.x_train, self.x_train, epochs=epoch, batch_size=batch,
shuffle=True, validation_data=(self.x_test, self.x_test))
```

Autoencoder for labels

- The training is performed only on the most common labels: only the labels that appear in at least 10 different images
- The input (= desired output) data is a vector with length equals to the number of all labels. The i -th vector cell contains the score of the i -th label or 0 if the i -th label does not appear in the image
- An input vector of approx. 1100 labels is compressed into a vector of 50/100 values with almost 0% loss!!

Clustering with labels encoder

- How to test the performance of the encoder level?
 - Perform classification/clustering using as input data the encoder output
- I used sklearn and k-means algorithm on the n-th dimensional encoder output to perform unsupervised classification of the images

- The result is very good!
- Classes are not tagged but we can visually recognize that that we have a good classification

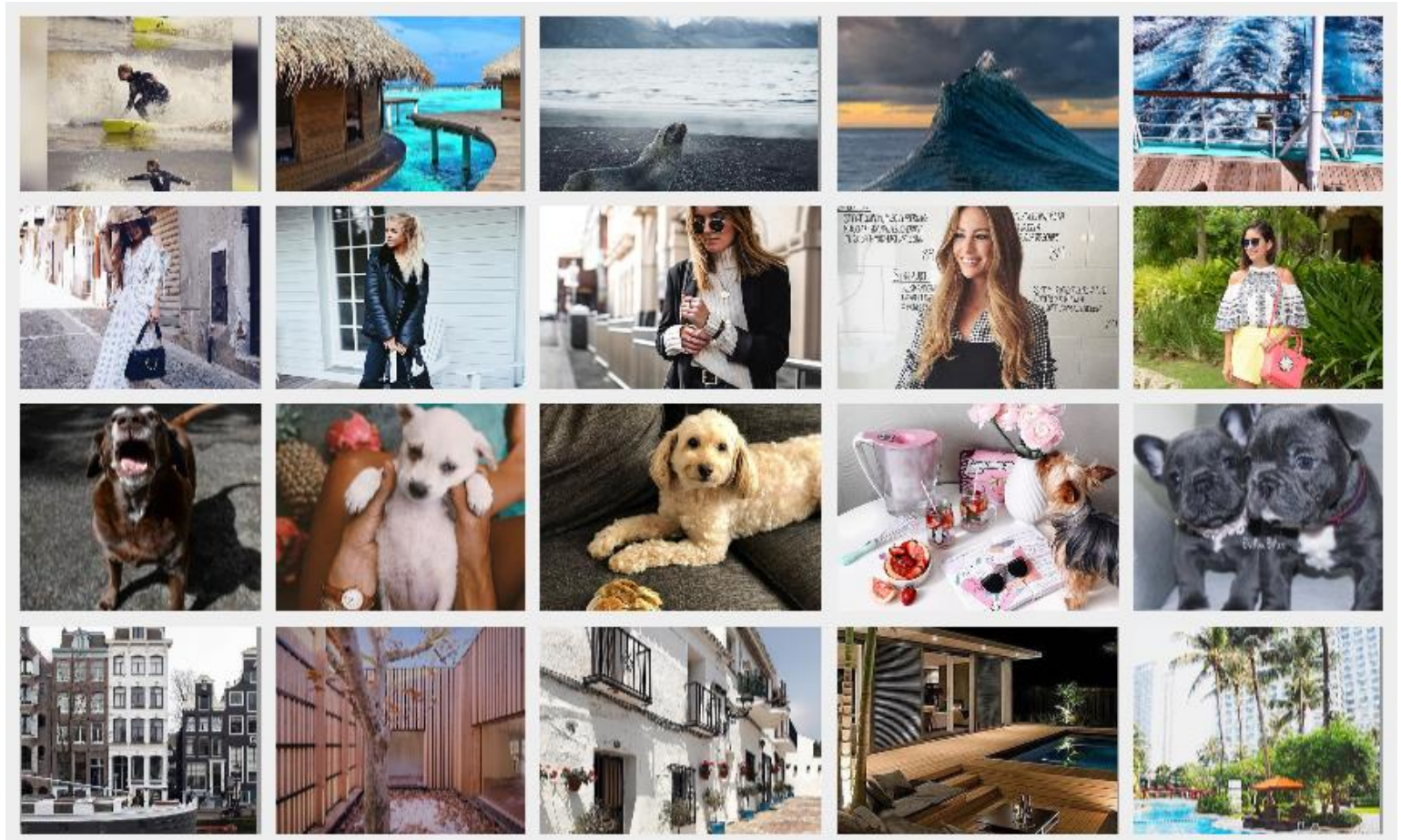
```
labelTrainer = LabelTrainer()
labelTrainer.read_features(filename = 'gvision.json', common_labels_min = 10)
labelTrainer.build_model(encoding_dim=100)
labelTrainer.train(epch=30)

print('Building matrix...')
X = np.empty((0, labelTrainer.encoding_dim))
for filename in labelTrainer filenames:
    _, encodedLabel = labelTrainer.get_data(filename)
    X = np.vstack([X, encodedLabel.tolist()])

print('Clustering...')
kmeans = KMeans(n_clusters=50, random_state=0).fit(X)
predictions = kmeans.predict(X)
```

Clustering with labels encoder

- Each row contains 5 random images from a specific cluster



Autoencoder for labels

- How can we use the encoder for like prediction?
- If the mean number of likes is such a good attribute
 - Then we can ignore other meta-data (such as number of followers, comments, hashtags)
 - For each image I compute 'number of likes' / 'mean number of user likes' and I call it '**positivity**'.
 - It's a value > 1 for 'good images' (many likes) and < 1 for 'bad images' (less likes than usual)
- The new task is to predict the positivity value. If computed correctly, then it's sufficient to multiply it for the mean number of likes to obtain the predicted number of likes

Third approach: labels

- Prediction of the positivity value using the encoded output of the label autoencoder
- Again, a very simple network!

```
def train_labels():  
    model.add(Dense(200, input_dim=positivitymatrix.shape[1], activation='relu', activity_regularizer=regularizers.l1(0.01)))  
    model.add(Dropout(0.2))  
    model.add(Dense(100, activation='relu', activity_regularizer=regularizers.l1(0.01)))  
    model.add(Dense(50, activation='relu'))  
    model.add(Dense(1))  
  
    model.compile(loss='mape', optimizer='adam', metrics=['mae', 'mse', 'mape'])  
    model.fit(pos_x_train, pos_y_train, epochs=5000, batch_size=8, validation_data=(pos_x_test, pos_y_test))
```

Third approach: labels

- The mape is now 34% on the test data 😊
- This proves that some class of images tend to gain more likes than others
- Why the network can't go further?
 - It's not unusual that a user publishes posts about the same arguments
 - If the labels are the same and the number of likes are very different, the network can't learn anything

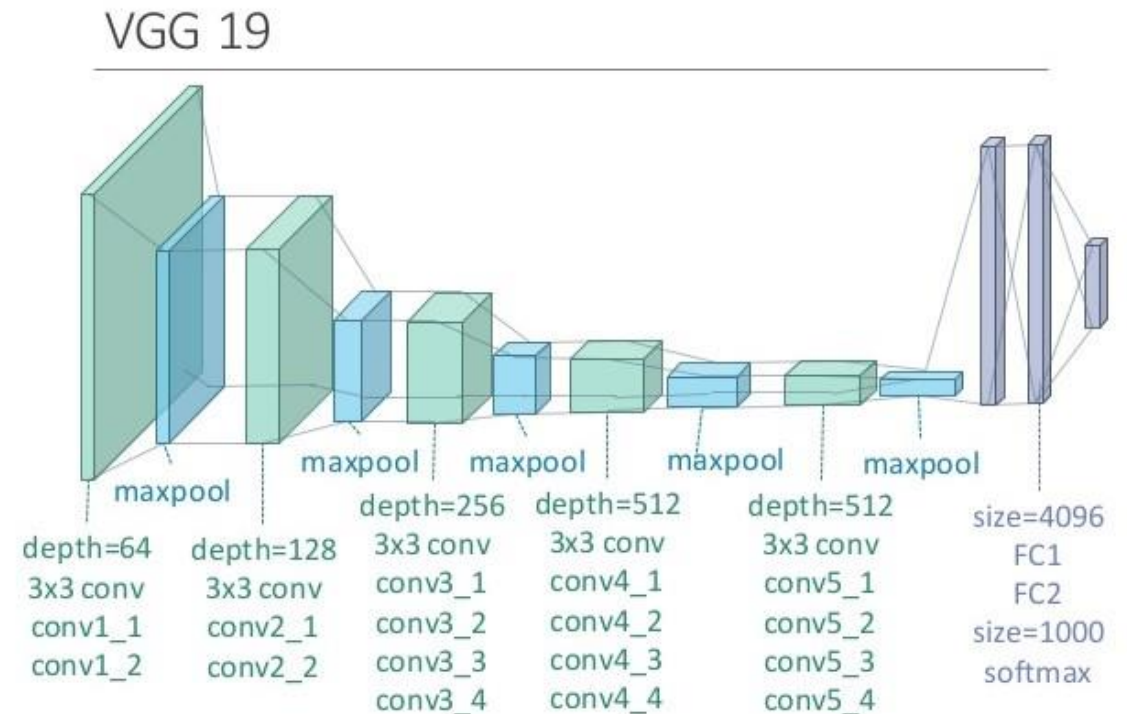
Fourth approach: CNN to predict positivity

- I built a new (larger) CNN to predict the positivity value, hoping to avoid the strong dependence between number of likes and mean number of likes
- The network does not fit into the GPU memory: long training 😞
- Good result: 36% mape. It seems like a CNN-only solution is still possible 😊

```
def train_images2():
    model.add(Conv2D(96, (11,11), strides=4, input_shape=(IMAGE_SIZE,IMAGE_SIZE,CHANNELS), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=3, strides=2))
    model.add(BatchNormalization())
    model.add(Conv2D(256, (5,5), strides=2, activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=3, strides=2))
    model.add(BatchNormalization())
    model.add(Conv2D(384, (3,3), activation='relu'))
    model.add(Conv2D(256, (3,3), activation='relu'))
    model.add(MaxPooling2D(pool_size=3, strides=2))
    model.add(Flatten())
    model.add(Dense(4096, activation='relu'))
    model.add(Dense(4096, activation='relu'))
    model.add(Dense(1))|
```


CNN to predict positivity: VGG-19

- To achieve big results you need a big CNN
- Keras has the 'most famous' CNNs pre-installed and pre-trained (just need to download the weights file the first time)
- VGG-19 (Oxford) is a 19-layer CNN trained on the ImageNet 2014 database that achieved ground breaking results. It's available on Keras
- If we remove the last layer, we have 4096 features for each image
- I used the 4096 features as input for a simple NN (only dense layers) to predict the positivity value
- The mape is still 36/37% ☹️



Fifth approach: cluster prediction

- As said before, labels can not predict well if the user always posts the same type of images, getting low and high number of likes for the same labels
 - We need a metric for each user to measure if a certain image type is 'surely good', 'surely bad', or 'unknown'
 - We can do that with clustering!
- Given a value $a > 0$:
 - For each cluster we compute the 'performance' of the cluster for the specific user using the mean of the positivity values of the user images belonging to that specific cluster
 - If the mean is $> a$, then the performance is 1 (good cluster)
 - If the mean is $< -a$, then the performance is -1 (bad cluster)
 - Otherwise, the performance is 0 (unknown)
- To avoid inserting too many dependency, this computation is made on only half the user images (randomly)

Fifth approach: cluster prediction

- We now have a value for each image that tells us if, for the specific user, it is a good or a bad image (or uncertainty)
- To obtain this value, we first have to find the cluster the image belongs to, then we take the performance of the cluster of the user that posted the image
- This new value is appended to the encoded label vector of the third approach. The whole vector is used as the input of the same network
- The mape drops to 30% 😊
- We achieved a good result, but at the cost of helping our network with information that strongly depends with the output itself 😞

Conclusion

- Predicting Instagram likes is a very hard task. Predictions with great accuracy might be impossible due to great variance of the number of likes
- The metric that most influences the number of likes of the next image is the average number of likes of the previous images
- There are classes of images that are in general more likely to have a lot of likes (user-independent)
- Convolutional Neural Networks can find patterns to help predictions
- Information based on a specific user about what classes of images are more likely to obtain a high number of likes can help predictions (user-dependent)

Conclusion

- Dataset, download scripts and DL source code are available at <https://github.com/riccardoalbertazzi/instagramlikeprediction>

THANKS FOR THE ATTENTION