LightHouse

COMS6998_2017Spring_report
Yilan Ji (yj2425), Haoyan Min (hm2689),
Guowei Xu (gx2127), Jiayang Li (jl4305)
Columbia University

1. Overview

LightHouse is a webapp that helps users to explore interesting photos of this world and share their opinions with each other. We first present users some photos from Flickr API, and then collect their preference tags and run machine learning algorithm perceptron, which involves training and predicting on Spark and passing messages with AWS SQS service. Based on the result, we will gather some photos that might have a higher probability to please the users, and then invite users to explore the personalized collection of photos for them. We also implement a general channel chatroom for all users where they can share their opinions about the photos or even find people that interest them and have further interactions.

2. Architecture

As shown in Figure 1, we utilize MongoDB as a remote server to store user's information, use SQS to handle data transportation between web server and Spark. A complete user preference customizing Machine Learning cycle is illustrated in Figure 1: Once the user requests for everyday pictures, we fetch the pictures from Clarifai API, display them, user gives feedback of the picture while viewing, the server sends the preference feedback as well as the tags string to Spark through SQS to update the ML model. When the user asks for recommendation, the server sends a SQS message containing user ID, then Spark fetches a list of images by Pixabay API, filters the list by the model of unique user ID received from server, and sends the list to server by SQS. The server responds the user's request once receiving the message from Spark.

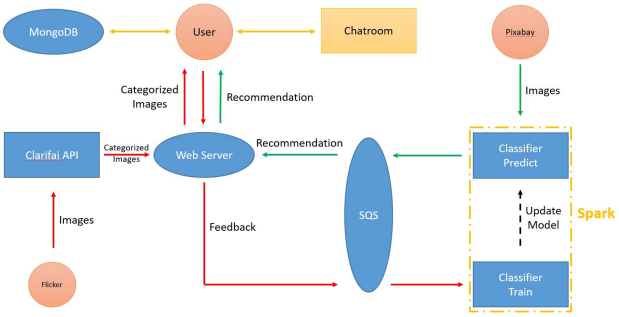
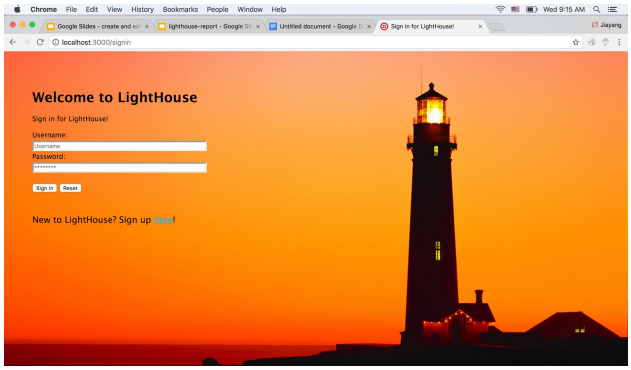
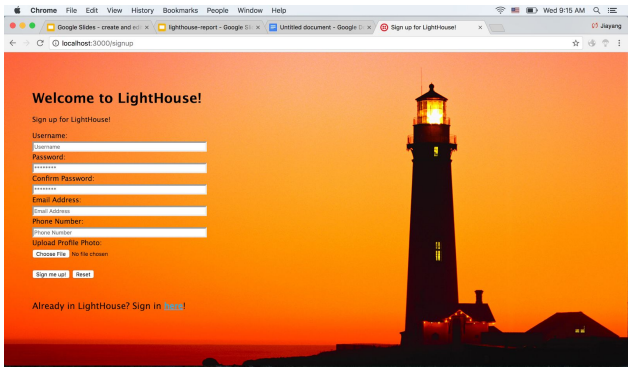


Figure 1

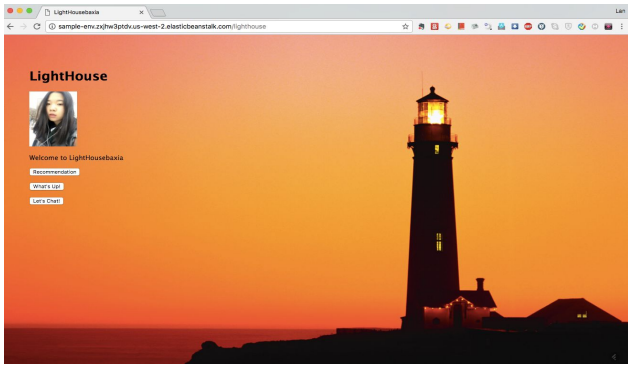
3. Screen Shot

Below you will see that we have a good UI.
Signin/signup page:

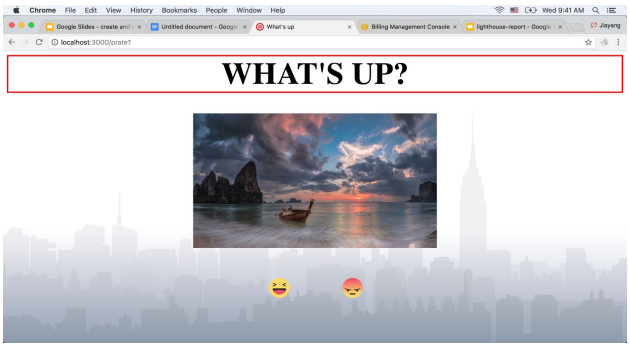


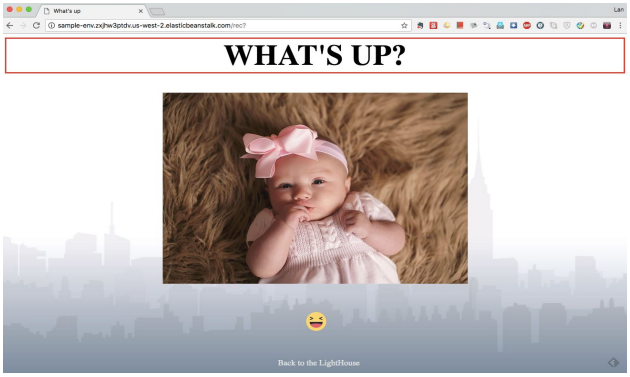


Homepage:

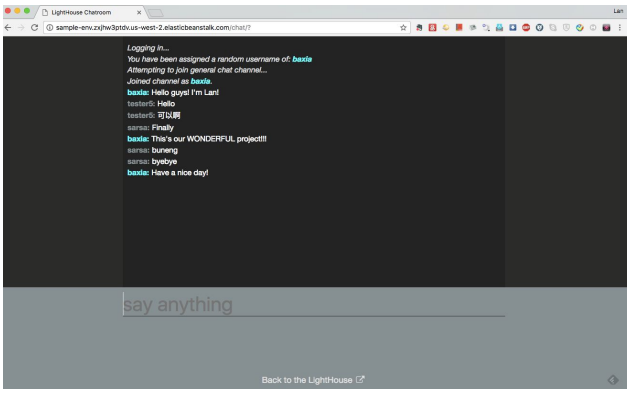


What's up page:

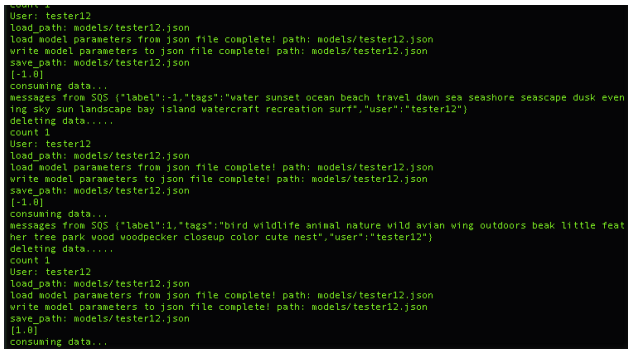




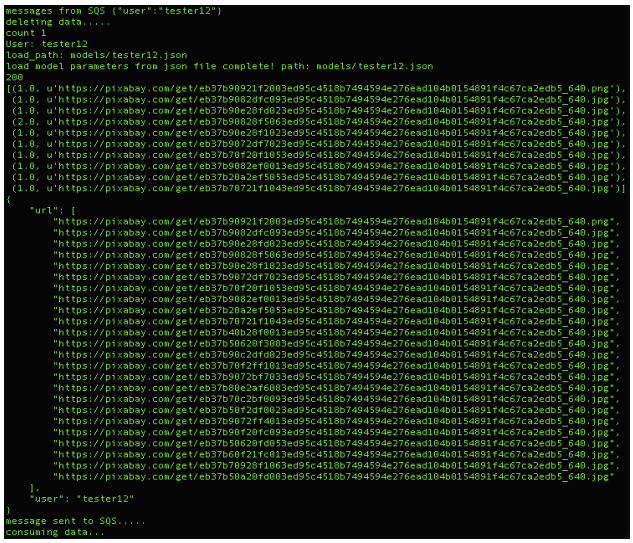
Chatroom page:



Screenshot for training feedback models:



Screenshot for recommendations:



4. Code Design

4.1 Session Control

We use Express.session to realize session control. If a get request contains no username in session, we will redirect the frontend to signin.

```
function restrict(req, res, next) {
  if (req.session.user) {
    next();
  } else {
    req.session.error = 'Access denied!';
    res.redirect('/signin');
  }
}
```

4.2 Sign up(/signup)

Once receiving request for signing up, the server will first check whether the username is used, if not, insert the user information to DB and redirect the link to signin.

4.3 Sign in(/signin)

Once receiving request for signing in, the server will check the existence of username and correctness of password, if both are correct, the server would maintain a session for the user and store the user information in req.session, then redirect the frontend to home page.

4.4 Homepage(/lighthouse)

The homepage displays the user image and has three buttons directing to Recommendation, What's up(Training Machine Learning Model), and Chatroom.

4.5 What's up(/prate)

This page displays everyday image from Flicker, collects user preference feedback, gets picture content via Clarifai API and finally sends the feedback, username and picture tags to Spark by SQS.

```
router.get('/feedback', function(req, res){
  var feedback=req.query.f;
  var pic_url = req.query.pic_url;
  console.log(typeof feedback);
  console.log(pic_url);
  console.log(req.session.user);
  var db=req.db;
  var collection=db.get('userfeedback');
  var labels='';
  Clarifai_app.models.predict(Clarifai.GENERAL_MODEL, pic_url).then(
    function(response) {
      for(var i=0;i<response.outputs[0].data.concepts.length;i++){
        if (response.outputs[0].data.concepts[i].name != 'no person'){
          labels=labels+response.outputs[0].data.concepts[i].name+' ';
          //console.log(response.outputs[0].data.concepts[i].name);
        }
      }
      labels=labels.substring(0,labels.length-1);
      console.log(labels);
//sqs
      var dic={
        "label":Number(feedback),
        "tags":labels,
        "user":req.session.user
      };
      console.log(dic);
      var dic_2=JSON.stringify(dic);
      console.log(dic_2);
      var params={
        MessageBody: dic_2,
        QueueUrl: 'https://sqs.us-west-2.amazonaws.com/145842502534/nofeedback',
      };
      sqs.sendMessage(params, function(err, data) {
        if (err) console.log(err, err.stack); // an error occurred
        else     console.log(data);           // successful response
      });
    },
    function(err) {
       console.error(err);
    });
});
```

4.6 Recommendation(/rec)

When user requests for recommendation, we firstly send a SQS message containing username to tell the Spark to use the corresponding model, then continue receiving message until the Spark returns a SQS message with a list of urls of filtered images.

```
router.get('/recimg',function(req,res){
  //sqs
  var u=JSON.stringify({'user':req.session.user});
  var params={
    MessageBody: u,
    QueueUrl: 'https://sqs.us-west-2.amazonaws.com/145842502534/lighthouseusername',
  };

  sqs.sendMessage(params, function(err, data) {
    if (err) console.log(err, err.stack); // an error occurred
    else     console.log(data);           // successful response
  });
  recur(req,res);
});
```

```
function recur(req,res){
  var receive={
    QueueUrl: 'https://sqs.us-west-2.amazonaws.com/145842502534/sparkfeedback',
    WaitTimeSeconds: 2
  };
  sqs.receiveMessage(receive, function(err, data) {
    if (err) console.log(err,err.stack); // an error occurred
    else{
      console.log('receiving');
      console.log(data);
      if(data!=undefined && data['Messages']!=undefined && data['Messages'][0]!=undefined && data['Messages'][0]['Body'] !=undefined){
        console.log(data['Messages'][0]['Body']);
        var body=JSON.parse(data['Messages'][0]['Body']);
        console.log(body);
        if(body['user']==req.session.user){
          var del={
            QueueUrl: 'https://sqs.us-west-2.amazonaws.com/145842502534/sparkfeedback',
            ReceiptHandle: data['Messages'][0]['ReceiptHandle']
          };
          sqs.deleteMessage(del,function(err,data){
            if(err) console.log(err,err.stack);
            else{
              console.log(data);
              console.log('deleted');
              console.log(body['url'].length);
                console.log(body['url']);
                res.send(body['url']);
            }
          });
        }
        else{
          recur(req,res);
        }
      }
      else{
        recur(req,res);
      }
    }
  });
}
```

We design a function recur(req,res) to recursively receive message from SQS sent by Spark until receiving the message with the same username as the request's. The function reads the message, saves the wanted one's content, deletes it, and responds the frontend with the list fetched from SQS. The reason for recursion rather than iterating is to maintain the time order of callback function.

4.7 Chatroom

We actually tried two chatroom APIs. The second API we tried is Twilio's chat API. The way we connect the users is to create a general channel and let users join it by searching the keyword 'general'. The first user will thus create the channel and the following can find it and join it. Each channel has a capacity of 100 people so if there are already 100 people registered in this chatroom, new user will fail to join the channel. Also, username of our webapp, which is retrieved via the session environment, is passed into the the usertoken of Twilio in app.js.

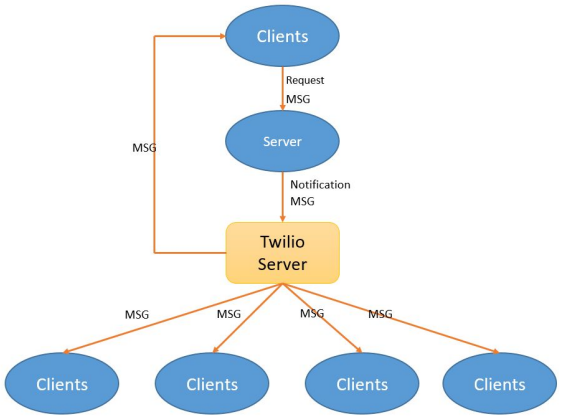
We implement a chatroom for users to network with each other. Twilio API is the service we use to establish the connection between the server and the clients. When a user issues a new request, such as login or a new message to be sent, the front end detects it and sends a notification to the Twilio server. Corresponding events are then directed to all the clients (all users that are in the chatroom) so that everyone is synchronized on this event. Specifically there are 4 types of notifications/events:

1). When a new user connects to our server, he will emit an event called newUser and the server will emit an event called newConnection with a list of all participants to all connected clients.

2). When a client disconnects from the server, an event called disconnect is automatically captured by the server. It will then emit an event to all participants with the id of the client that disconnected.

3). When a client sends a message through our POST method, the server will emit an event called incomingMessage which will send the sender's name and the message to all clients to show on their screens.

ChatRoom Architecture:



Connection between the frontend server and the Twilio server: user login event, user disconnect event.

```
io.on("connection", function(socket){
  /*
   When a new user connects to our server, we expect an event called "newUser"
   and then we'll emit an event called "newConnection" with a list of all
   participants to all connected clients
   */
  socket.on("newUser", function(data) {
    participants.push({id: data.id, name: data.name});
    io.sockets.emit("newConnection", {participants: participants});
  });

  /*
   When a client disconnects from the server, the event "disconnect" is automatically
   captured by the server. It will then emit an event called "userDisconnected" to
   all participants with the id of the client that disconnected
   */
  socket.on("disconnect", function() {
    participants = _.without(participants,_.findWhere(participants, {id: socket.id}));
    io.sockets.emit("userDisconnected", {id: socket.id, sender:"system"});
  });
});

//Start the http server at port and IP defined before
http.listen(app.get("port"), app.get("ipaddr"), function() {
  console.log("Server up and running. Go to http://" + app.get("ipaddr") + ":" + app.get("port"));
});
```

IncomingMessage event.

```
//POST method to create a chat message
app.post("/message", function(request, response) {
  //The request body expects a param named "message"
  var message = request.body.message;
  messages.push(message)
  console.log(messages)
  //If the message is empty or wasn't sent it's a bad request
  if(_.isUndefined(message) || _.isEmpty(message.trim())) {
    return response.json(400, {error: "Message is invalid"});
  }
  //We also expect the sender's name with the message
  var name = request.body.name;
  //Let our chatroom know there was a new message
  io.sockets.emit("incomingMessage", {message: message, name: name});
  //Looks good, let the client know
  response.json(200, {message: "Message received"});
});
app.get("/history", function(request, response) {
  var msgHistory=messages;
  //Render the view called "index"
  response.render("index");
});
```

4.8 Machine Learning ------ Perceptron on Spark:

The perceptron is a classic learning algorithm for the neural model of learning. Like K-nearest neighbors, it is one of those machine learning algorithms that is simple and yet works amazingly well for some types of problems. It's actually quite different than either the decision tree algorithm or the KNN algorithm, because perceptron is online and error-driven. We design two basic perceptron algorithms: Online Perceptron and Average Online Perceptron based on algorithms provided in *A Course in Machine Learning by Hal Daumé III*. First, extracting features from "raw" text data by HashingTF() function in Spark MLlib. HashingTF() is a Transformer which takes sets of terms and converts those sets into fixed-length feature vectors. Second, using those sparse vectors as input, 1 or -1 as label representing the binary class. Third, user can choose to train which model, set parameters including iterations and initial weight vector. Finally, predicting the label of test data utilizing the trained model.

It is crucial to determine the parameters of classifiers to get better accuracy and minimal error rate. Therefore, we compare the performance of the models with different parameters(average or not, number of features, iterations) with offline models provided in Spark MLlib, e.g. NaiveBayes;SVMwithSGD;LogisticRegressionWithLBFGS. Data for comparison comes from restaurant reviews {text, label} (50000 as training data; 10000 as test data) whose labels are implying the positive or negative evaluation for pictures from users. Results are as shown in terminal screen shot below. In our application, we choose AveragePerceptron with iterations equal to 10, number of features equal to 1000 to learn user preference.

```
Test error rate of SVMWithSGD Model(iterations=100):  0.227
Test error rate of LogisticRegressionWithLBFGS Model:  0.1489
Test error rate of OnlinePerceptron(iterations=1):  0.2413
Test error rate of OnlinePerceptron(iterations=10):  0.2413
Test error rate of AveragePerceptron(iterations=1):  0.1517
Test error rate of AveragePerceptron(iterations=1) after single-pass OnlinePerceptron:  0.1479
Test error rate of AveragePerceptron(iterations=10) after 10 time pass OnlinePerceptron:  0.1455
Test error rate of AveragePerceptron(iterations=5)::  0.1451
Test error rate of AveragePerceptron(iterations=10):  0.1432
Test error rate of AveragePerceptron(iterations=100):  0.1487


Confusion Matrix of models:
+--------------------------+---------+-------------+-------------+-------------+-------------+
|Model                     |errorRate|falseNegative|falsePositive|trueNegative|truePositive|
+--------------------------+---------+-------------+-------------+-------------+-------------+
|NaiveBayes                |0.1695   |712          |983          |2301         |5924         |
|SVMWithSGDItr100          |0.227    |248          |2022         |1342         |6388         |
|LogisticRegressionWithLBFGS|0.1489   |655          |834          |2530         |5981         |
|OnlinePerceptronItr1      |0.2413   |2003         |483          |2961         |4633         |
|OnlinePerceptronItr10     |0.2413   |2092         |338          |3026         |4544         |
|AveragePerceptronItr1     |0.1517   |645          |872          |2492         |5991         |
|AveragePerceptronOPItr1   |0.1479   |664          |815          |2549         |5972         |
|AveragePerceptronOPItr10  |0.1455   |657          |798          |2566         |5979         |
|AveragePerceptronItr5     |0.1451   |642          |809          |2555         |5994         |
|AveragePerceptronItr10    |0.1432   |632          |800          |2564         |6004         |
|AveragePerceptronItr100   |0.1487   |684          |803          |2561         |5952         |
+--------------------------+---------+-------------+-------------+-------------+-------------+
```

Our model use online classification model so that we do not to need store any picture but only model attributes of each user. Besides, we also deploy our spark models on AWS EMR cluster to run the overall web app on cloud.

Perceptron model class:

Initialize weight vectors and average weights history.

```
class PerceptronforRDD():
    def __init__(self,numFeatures=1000, w=np.zeros(1000),b=0.0,u_avg=np.zeros(1000),beta_avg=0.0,count_avg =1.0):
        if len(w)!= numFeatures:
            self.w = np.zeros(numFeatures)
            self.u_avg = np.zeros(numFeatures)
        else:
            self.w = w
            self.u_avg =u_avg
        self.b = b
        self.beta_avg = beta_avg
        self.count_avg = count_avg
```

Main functions: train and predict

Predict using map function

```
    def predict(self,data):
        w = self.w
        b = self.b
        predict = data.map(lambda x: x.dot(w)+b)
        predict = predict.map(lambda p: -1.0*(p<0)+1.0*(p>=0))
        return predict
```

Training is limited by data input sequence, model update sequentially.

```
    def AveragePerceptron(self, data, label, MaxItr=10):
        label = label.map(lambda x: -1.0*(x==0.0 or x==-1.0)+(x==1.0))
        label = label.collect()
        data = data.collect()
        ind = range(len(data))
        for time in range(MaxItr):
            random.shuffle(ind)
            for i in ind:
                pred = data[i].dot(self.w) + self.b
                if label[i]*pred<=0:
                    self.w = self.w + label[i]*data[i].toArray()
                    self.b = self.b + label[i]
                    self.u_avg = self.u_avg + self.count_avg*label[i]*data[i].toArray()
                    self.beta_avg = self.beta_avg + self.count_avg*label[i]
            self.count_avg += 1.0
        self.w = self.w - self.u_avg/self.count_avg
        self.b = self.b - self.beta_avg/self.count_avg
        return [self.w,self.b]
```

Training Worker:

Consume message{“user”:user,”tags”:tags”,”label”:label} from SQS “userfeedback” queue, train a perceptron model for every user online. Instead of storing tags data, we only need to update and save the model for every user.

```
    def WorkonMessages(self, messages,sc):
        try:
            message = messages[0]
            user = message["user"]
            print "User:",user
            model = PerceptronforRDD(numFeatures=1000)
            try:
                print "load_path:","models/"+user+".json"
                model.load("models/"+user+".json",average=True)
            except Exception,e:
                pass
            messages = sc.parallelize(messages)
            tags = messages.map(lambda x: x["tags"].split(" "),preservesPartitioning=True)
            labels = messages.map(lambda x: x["label"],preservesPartitioning=True)
            tf = self.hashingTF.transform(tags)
            model.perceptronBatch(data=tf,labels=labels,MaxItr=10)
            model.save("models/"+user+".json",average=True)
            print "save_path:","models/"+user+".json"
            predict = model.predict(tf)
            print predict.collect() # Show the predict results on terminal and compare with true labels
```

Predict Worker:

Consume message{“user”:user} from SQS “userrecomendation” queue, request latest/popular pictures from Pixabay API, predict and send positive predictions to “sparkfeedback” queue.

```
def WorkonMessages(self, messages,sc):
    try:
        message = messages[0]
        user = message["user"]
        print "User:",user
        model = PerceptronforRDD(numFeatures=1000)
        try:
            print "load_path:","models/"+user+".json"
            model.load("models/"+user+".json",average=True)
        except Exception,e:
            pass
        response = requests.get("https://pixabay.com/api/?key=5321184-68d88d45c3c32345d8c6768c7&image_type=photo&order=popular&per_page=200").json()
        results = response["hits"]
        print len(results)
        doc=[]
        for i in range(200):
            result = results[i]
            doc.append({"tags":result["tags"].split(", "),"url":result["webformatURL"]})
        messages = sc.parallelize(doc)
        tags = messages.map(lambda x: x["tags"],preservesPartitioning=True)
        links = messages.map(lambda x: x["url"],preservesPartitioning=True)
        tf = self.hashingTF.transform(tags)
        predict = model.predictPositive(tf,links)
        prediction = predict.map(lambda x:x[1]).collect()
        positive = {"user":user,"url":prediction}
        print json.dumps(positive, indent=4)
        queueName = 'sparkfeedback'
        queue = self.sqs.get_queue_by_name(QueueName=queueName)
        queue.send_message(MessageBody=json.dumps(positive))
        print "message sent to SQS...."
```

5. Conclusion

In this project, we build a web app called LightHouse to present users a set of interesting pictures. Along the process LightHouse collects users’ feedback on the pictures and uses machine learning algorithm perceptron to analyze users’ preference on some categories over the other. Then a user profile is created to recommend new pictures that mostly s(he) will enjoy. Besides, chatroom is where people can share their ideas about what they see in our app. The final result is satisfying as you will see in our demo, if a user really likes birds and gives a smiling face to all the bird pictures, then our recommendation system will present mostly bird or similar animal photos. This concludes that LightHouse is smart and fun photo recommendation app.

6. Future Work

We do a decent job implementing the user signin/signup, the learning/recommendation system and the chatroom. However, several progresses can still be made:

1). Support changing password in signin/signup. From a perspective of practical use, this is a vital functionality and is not hard.

2). Create multiple chatroom based on users’ feedback. For the moment we have a general chatroom where every user can enters. A better way would be to create multiple chatrooms each associated with a topic and all users in a chatroom are predicted to like the topic.