ML Capstone Required Tasks

Required Tasks

Task 1: Project Setup

1. Instantiated a AWS AMI GPU Instance.
2. Tensorflow and Keras is preinstalled on the GPU Instance provided by AWS
3. Installed prerequisites for Flask in virtual Environment with Tensorflow Backend
4. Setup a WebApp using Flask
5. Configure the User Database

Requirements:

```
alembic==1.0.0
click==6.7
dominate==2.3.1
Flask==1.0.2
Flask-Bootstrap==3.3.7.1
Flask-JWT-Extended==3.13.0
Flask-Migrate==2.2.1
Flask-SQLAlchemy==2.3.2
itsdangerous==0.24
Jinja2==2.10
Mako==1.0.7
MarkupSafe==1.0
PyJWT==1.6.4
python-dateutil==2.7.3
python-dotenv==0.9.1
python-editor==1.0.3
six==1.11.0
SQLAlchemy==1.2.12
visitor==0.1.3
Werkzeug==0.14.1

```

Task 2: Code the Service

1. used the inception resnet pre-trained to extract bottle neck features from my training data
2. next, used the output from the inception restnet as training input to my_model. those output from the pre-trained model is basically the bottle-neck features, edge detection and all. essentially, the pre-trained model does the heavy lifting
3. So whenever there is a new image to classify, first we need to extract the bottle-neck features from that image using the same pre-trained model
4. Implement the classify Endpoint for classification of skin cancer

```python
@app.route('/classify', methods=['POST'])
# @jwt_required
def classify():
    """
        It accepts Image for classification and returns the result
    """
    # Get test image ready
    try:
        test = request.files['file']
        test_image = path_to_tensor(test)
        print (test_image.shape)
        transfer_pred = make_transfer_prediction(test_image)
        print(transfer_pred.shape)
        prediction = make_custom_prediction(transfer_pred)
        print(prediction)
        predicted, prediction_stats = humanize_prediction(prediction)
        print(predicted, prediction_stats)
        return jsonify({
            'predicted class': predicted,
            'prediction stats': prediction_stats
            })
    except:
        raise
#         return "Could not make prediction"
```

Task 3: Do Register/Login

1. Implement the Register User Function using Security with API Token
2. Implement Login Functionality

```python
@app.route('/register', methods=['POST'])
def register():
    """
        This is the registration function, all fields - username, email password1
        and password2 are required. If registration is successful, a token is
        sent to the user for subsequent auths
    """

    if not request.is_json:
        return jsonify({"msg": "Missing JSON in request"}), 400
    data_is_valid = False
    msg = ''

    username = request.json.get('username', None)
    email = request.json.get('email', None)
    password1 = request.json.get('password1', None)
    password2 = request.json.get('password2', None)

    if username and email and password1 and password2:
        data_is_valid, msg = RegistrationValidator.registration_validator(username, email, password1, password2)
        if data_is_valid:
            # call create user object
            user = User(username=username, email=email)
            user.set_password(password1)
            user.save_to_db()
            access_token = AuthValidator.authenticate(username)
            return jsonify(access_token=access_token), 200
        else:
            return jsonify({"msg": msg}), 400
    else:
        return jsonify({"msg": "all fields are required"}), 400
```

```python
@app.route('/login', methods=['POST'])
def login():
    """
        function to generate tokens and return to user, it accepts valid username and password
    """
    if not request.is_json:
        return jsonify({"msg": "Missing JSON in request"}), 400

    username = request.json.get('username', None)
    password = request.json.get('password', None)
    if not username:
        return jsonify({"msg": "Missing username parameter"}), 400
    if not password:
        return jsonify({"msg": "Missing password parameter"}), 400

    # authenticate user here
    if AuthValidator.validate_user(username, password):
        access_token = AuthValidator.authenticate(username)
        return jsonify(access_token=access_token), 200
    else:
        return jsonify({"msg": "User can not be authenticated"}), 404
```

Task 4: Implement Error Handling

For all Eventualities Error Handling needs to be implemented:

- Incorrect Input : the application must act properly

Task 5: Test the WebApp

App must be tested from several Devices such as Tablets , smartPhones (on Emulator aswell real Devices). Check that Servcie works properly