

## 1 Déploiement d'un modèle de régression

Un exemple de déploiement d'un modèle de régression sur Heroku a été vu en cours. Le code complet de l'exemple est disponible ici :

[https://github.com/lezoray/Flask\\_Heroku](https://github.com/lezoray/Flask_Heroku)

et son déploiement ici sur Heroku et Render :

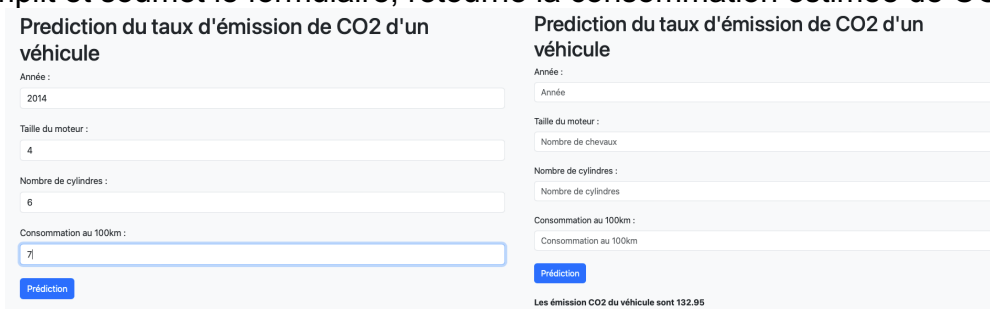
<https://insurance-app-7145f2499c96.herokuapp.com>

et

<https://insurance-app-kyfc.onrender.com>

Le code est récupérable en faisant un clone avec git en CLI ou bien en récupérant une archive. Inspirez-vous de ce qui a été fait dans cet exemple pour créer une application qui permet d'estimer le taux de CO2 d'un véhicule. Nous utiliserons une base de données nommée « FuelConsumption.csv » qui contient des attributs décrivant un véhicule et sa consommation en CO2. Créez, dans un fichier python nommé « mlmodel.py » (et non un notebook), un modèle de régression polynomiale (de degré 3) pour cette dernière à partir uniquement des attributs nommés 'MODELYEAR', 'ENGINE SIZE', 'CYLINDERS', 'FUELCONSUMPTION\_COMB'. Sauvegardez le modèle appris à l'aide de pickle dans un fichier nommé « model.pickle ».

Écrivez une application nommée « app.py » avec flask de manière à ce que cela déclenche un formulaire comme celui apparaissant à <https://co2-app-45bfdd69a66f.herokuapp.com> ou <https://co2app-ptqf.onrender.com/> et qui, lorsque l'on remplit et soumet le formulaire, retourne la consommation estimée de CO2 :



Testez votre application en local avec « python app.py » ou bien « gunicorn app:app ». Déployez ensuite cette application sur Render. (<https://www.render.com>). Créez au préalable les fichiers « requirements.txt » (avec `pipreqs` ou bien `pip freeze > requirements.txt`), « Procfile » et « runtime.txt ». Testez vos applications déployées pour vérifier qu'elles fonctionnent comme en local (Utilisez les valeurs [2014, 2, 4, 5] pour les données d'entrée).

## 2 Déploiement d'un modèle de classification d'images

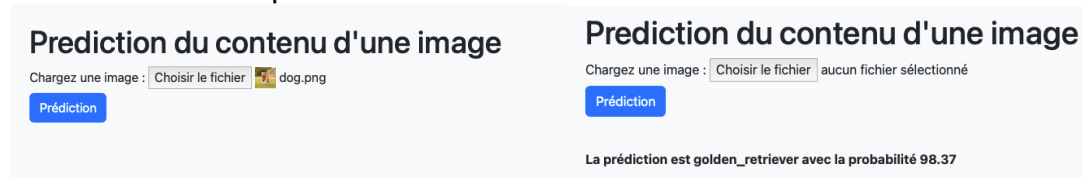
Dans le TP précédent nous avons utilisé VGG16 pré-entraîné sur ImageNet pour prédire le contenu d'une image. Cette fois nous allons reprendre le modèle ResNet50 pré-entraîné pour reconnaître le contenu d'une image parmi les classes d'ImageNet. En effet VGG16 est un modèle lourd dont l'ensemble des poids fait 500Mo, alors que ResNet prend cinq fois moins de place. Construisez une application qui demande à l'utilisateur de choisir une image via un formulaire. Cette image sera traitée par l'application afin d'afficher la classe trouvée par ResNet50. Inspirez-vous du code du TP1 avec VGG16, c'est le même principe, seul le réseau change.

En flask, pour récupérer une image via un formulaire, il faut :

1. Installer le package `flask-uploads` (ou `Flask-Reuploaded` si cela pose des problèmes de dépendances)
2. Importer des fonctionnalités de ce package : `from flask_uploads import UploadSet, configure_uploads, IMAGES`
3. Configurer le chargement comme s'effectuant dans le répertoire « `static/img` » (créez ce répertoire) :  

```
photos = UploadSet('photos', IMAGES)
app.config['UPLOADED_PHOTOS_DEST'] = './static/img'
configure_uploads(app, photos)
```
4. L'image sera alors automatiquement mise dans le répertoire précisé et son nom est récupérable par l'instruction `filename = photos.save(request.files['photo'])`

Exécutez votre application en local avec `gunicorn`. Vous devriez obtenir quelque chose comme ce qui suit :



## 3 Classification d'email

Nous allons à présent concevoir une application (interrogeable avec une api) qui permette de vérifier si un email est un spam ou non. Pour cela, deux modèles pré-entraînés vous sont fournis sur `ecampus`. Le premier permet de transformer un texte d'email en un vecteur de tokens et le second de prédire si un email est un spam ou non. Il vous faudra les charger (les modèles seront dans un répertoire nommé « `models` » :

```
cv = pickle.load(open("models/cv.pkl", 'rb'))
clf = pickle.load(open("models/clf.pkl", 'rb'))
```

et les utiliser pour classer un email :

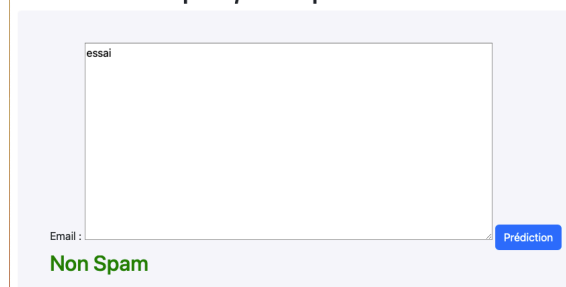
```
tokenized_email = cv.transform([email])
prediction = clf.predict(tokenized_email)
```

La prédiction vaudra plus ou moins un : +1 pour Spam et -1 pour Non Spam.

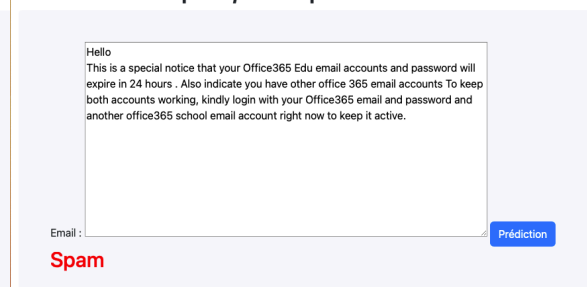
Écrivez une application flask qui permet de classer un email dont le contenu est saisi dans un formulaire et qui affiche si l'email est un spam ou non (dans le template html du formulaire, utilisez une instruction conditionnelle voir <https://jinja.palletsprojects.com/en/3.0.x/templates/#if>).

Voici des exemples d'utilisation de l'application en local (un exemple d'email spam est fourni sur ecampus) :

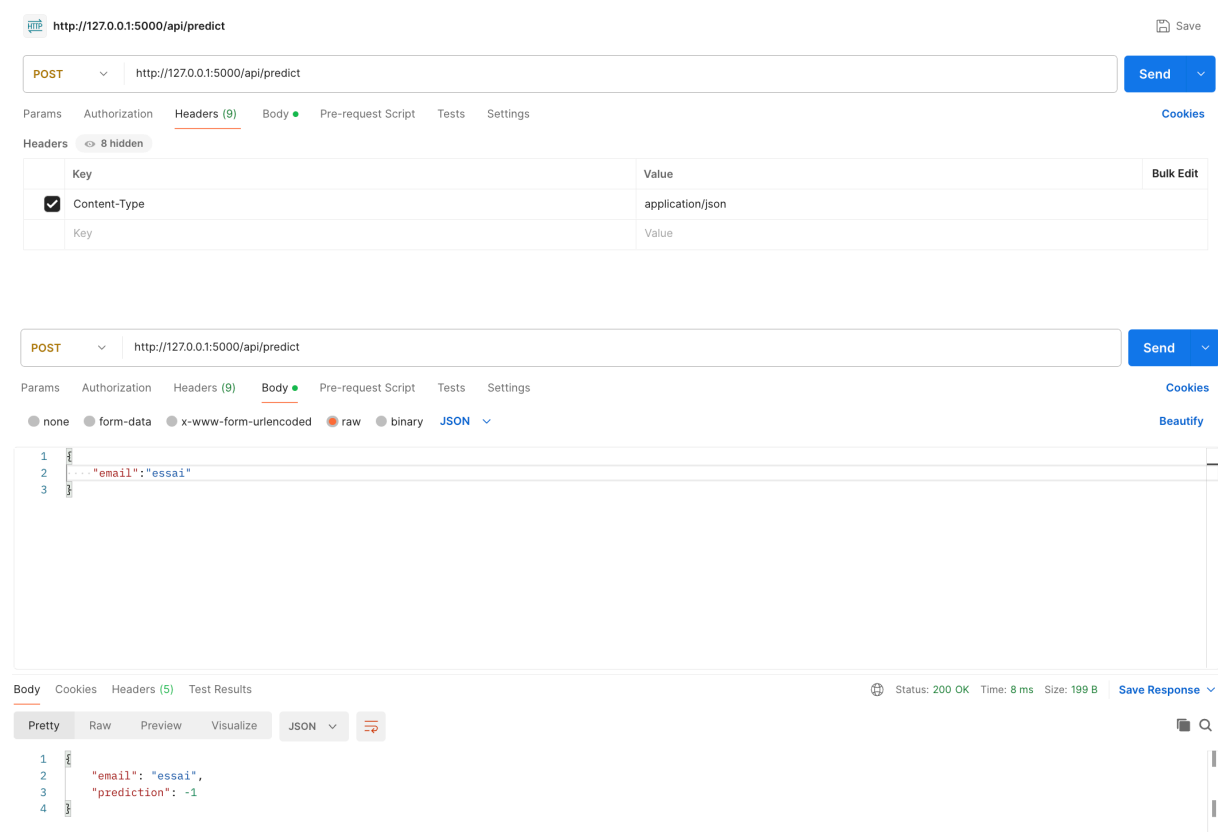
#### Prediction de spam/non-spam d'un email



#### Prediction de spam/non-spam d'un email



Ajoutez une méthode qui permette également d'utiliser l'application comme une API accessible à `'/api/predict'`. Cette API prendra des données envoyées en JSON et en POST et renverra le résultat en JSON. Les données seront alors récupérables avec l'instruction `data = request.get_json(force=True)`. Vous testerez votre API avec POSTMAN (suivez les captures ci-dessous pour configurer l'interrogation en JSON) :



The top screenshot shows the Postman interface for a POST request to `http://127.0.0.1:5000/api/predict`. The 'Headers' tab is active, showing a table with one header: `Content-Type` with value `application/json`. The 'Body' tab is also visible, showing a JSON payload: `{"email": "essai"}`.

The bottom screenshot shows the Postman interface for the same POST request. The 'Body' tab is active, showing the JSON payload: `{"email": "essai"}`. The status bar at the bottom indicates a successful response: `Status: 200 OK`, `Time: 8 ms`, `Size: 199 B`.

Pour finir déployez votre application sur Render. Un exemple est visible à : <https://spam-predict.onrender.com/>