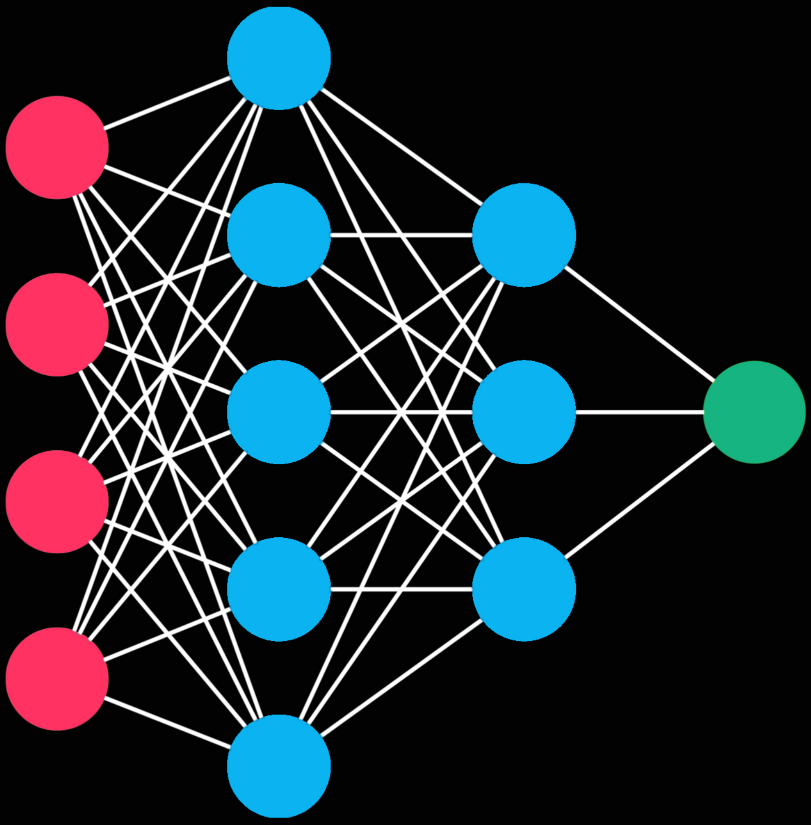
Venelin Valkov

Hacker's Guide to Machine Learning in Python



With Sklearn, TensorFlow 2 and Keras

Hacker's Guide to Machine Learning with Python

Hands-on guide to solving real-world Machine Learning problems with Scikit-Learn, TensorFlow 2, and Keras

Venelin Valkov

This book is for sale at http://leanpub.com/Hackers-Guide-to-Machine-Learning-with-Python

This version was published on 2020-07-13

Tweet This Book!

Please help Venelin Valkov by spreading the word about this book on Twitter!

The suggested hashtag for this book is #mlhackers.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#mlhackers

Contents

End to End Machine Learning Project

> TL;DR Step-by-step guide to build a Deep Neural Network model with Keras to predict Airbnb prices in NYC and deploy it as REST API using Flask

This guide will let you deploy a Machine Learning model starting from zero. Here are the steps you're going to cover:

- Define your goal
- Load data
- Data exploration
- Data preparation
- Build and evalute your model
- Save the model
- Build REST API
- Deploy to production

There is a lot to cover, but every step of the way will get you closer to deploying your model to the real-world. Let's begin!

Run the modeling code in your browser[1]

The complete project on GitHub[2]

Define objective/goal

Obviously, you need to know why you need a Machine Learning (ML) model in the first place. Knowing the objective gives you insights about:

- Is ML the right approach?
- What data do I need?
- What a "good model" will look like? What metrics can I use?
- How do I solve the problem right now? How accurate is the solution?
- How much is it going to cost to keep this model running?

In our example, we're trying to predict Airbnb[3] listing price per night in NYC. Our objective is clear - given some data, we want our model to predict how much will it cost to rent a certain property per night.

[1]https://colab.research.google.com/drive/1YxCmQb2YKh7VuQ_XgPXhEeIM3LpjV-mS
[2]https://github.com/curiousily/Deploy-Keras-Deep-Learning-Model-with-Flask
[3]https://www.airbnb.com/

Load data

The data comes from Airbnb Open Data and it is hosted on Kaggle[4]

> Since 2008, guests and hosts have used Airbnb to expand on traveling possibilities and present more unique, personalized way of experiencing the world. This dataset describes the listing activity and metrics in NYC, NY for 2019.

Setup

We'll start with a bunch of imports and setting a random seed for reproducibility:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.model_selection import train_test_split
import joblib

%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.5)

rcParams['figure.figsize'] = 16, 10

RANDOM_SEED = 42

np.random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)
```

Download the data from Google Drive with gdown:

```
!gdown --id 1aRXGcJlIkuC6uj1iLqzi9DQQS-3GPwM_ --output airbnb_nyc.csv
```

And load it into a Pandas DataFrame:

[4]https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data

```
df = pd.read_csv('airbnb_nyc.csv')
```

How can we understand what our data is all about?

Data exploration

This step is crucial. The goal is to get a better understanding of the data. You might be tempted to jumpstart the modeling process, but that would be suboptimal. Looking at large amounts of examples, looking for patterns and visualizing distributions will build your intuition about the data. That intuition will be helpful when modeling, imputing missing data and looking at outliers.

One easy way to start is to count the number of rows and columns in your dataset:

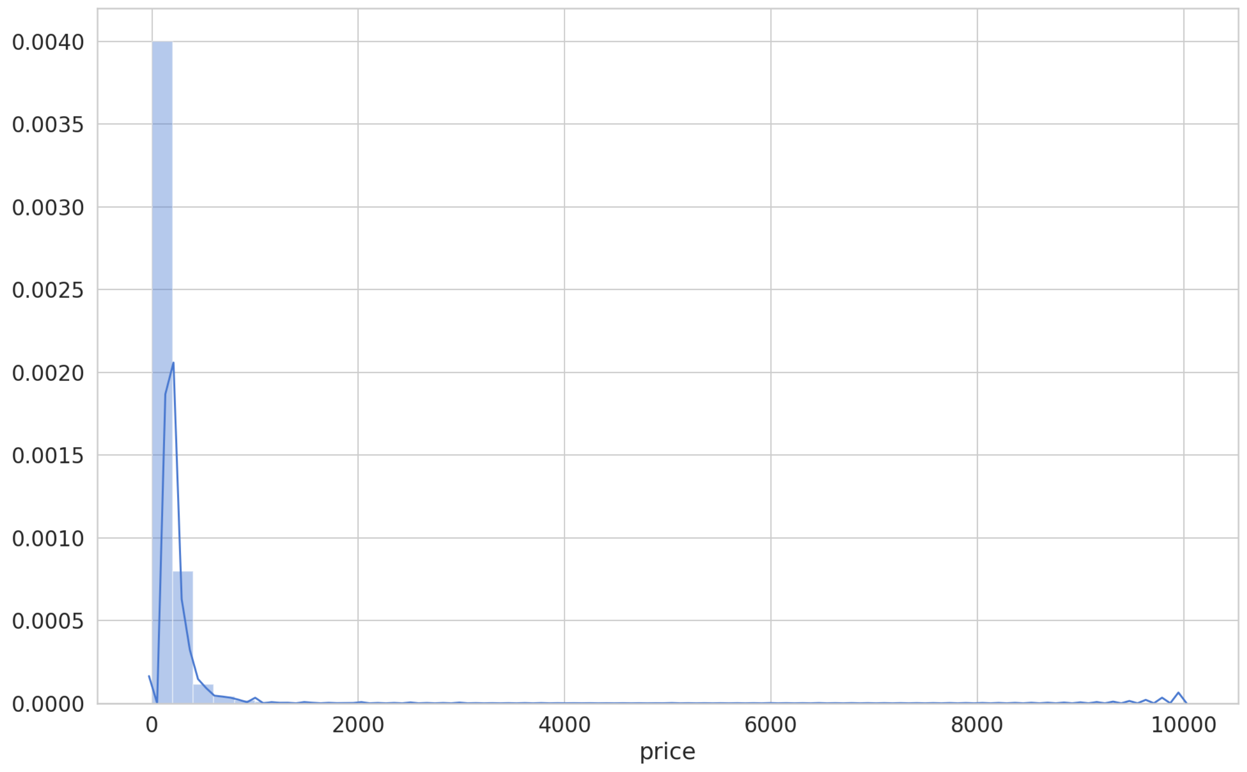
```
df.shape
```

```
(48895, 16)
```

We have *48,895* rows and *16* columns. Enough data to do something interesting.

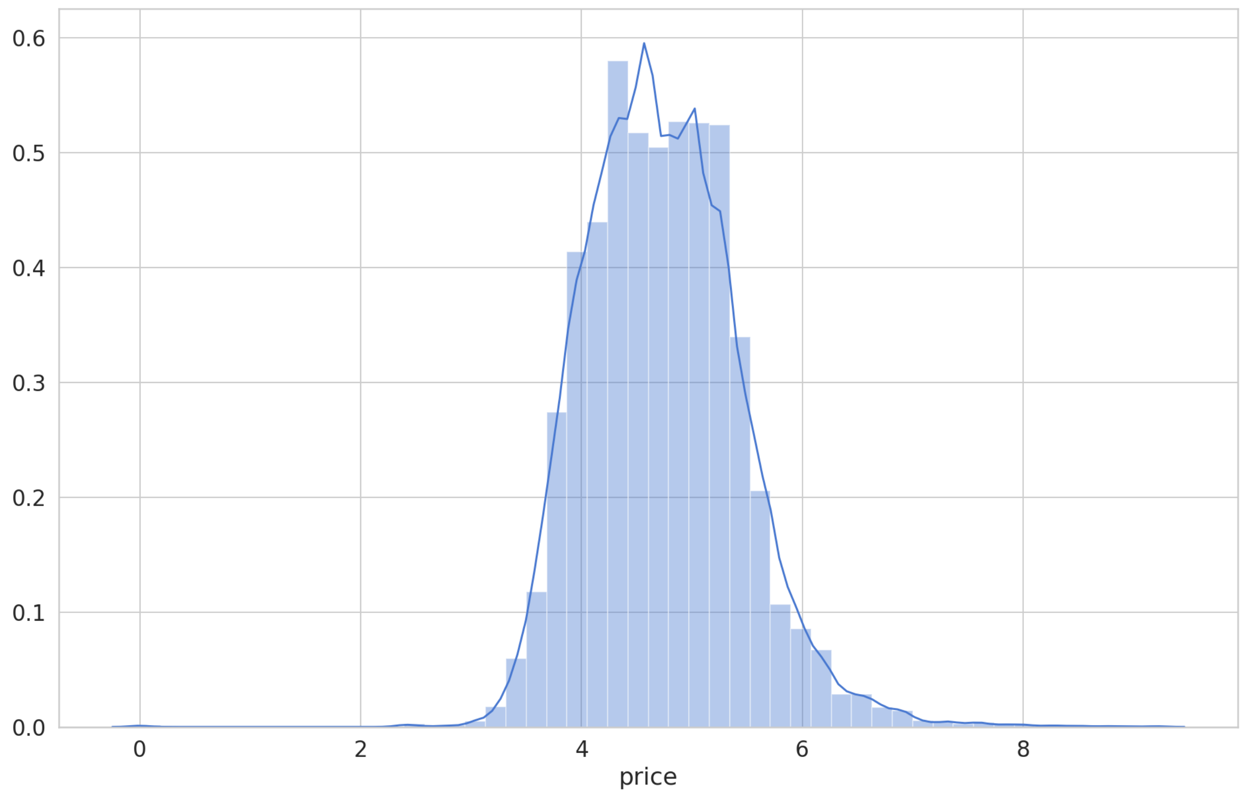
Let's start with the variable we're trying to predict `price`. To plot the distribution, we'll use `distplot()`:

```
sns.distplot(df.price)
```



We have a highly skewed distribution with some values in the 10,000 range (you might want to explore those). We'll use a trick - log transformation:

```
sns.distplot(np.log1p(df.price))
```

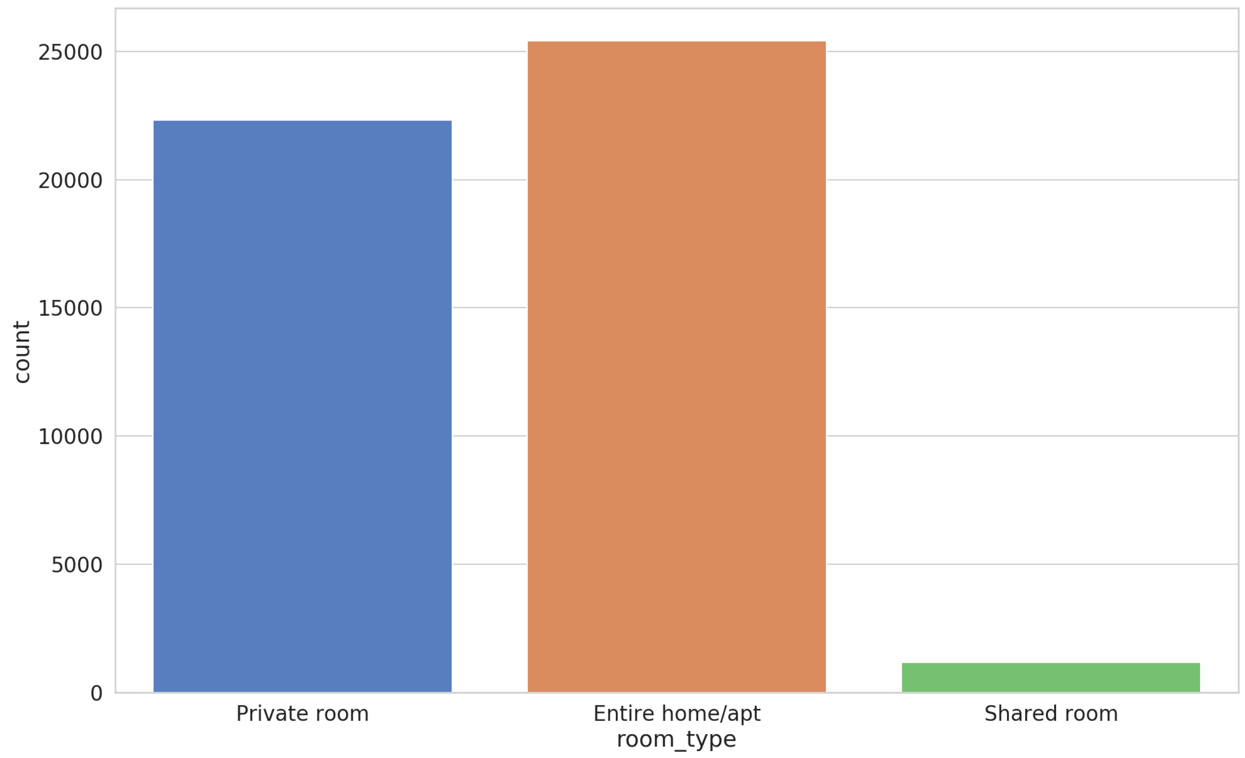


This looks more like a normal distribution. Turns out this might help your model better learn the data[5]. You'll have to remember to preprocess the data before training and predicting.

The type of room seems like another interesting point. Let's have a look:

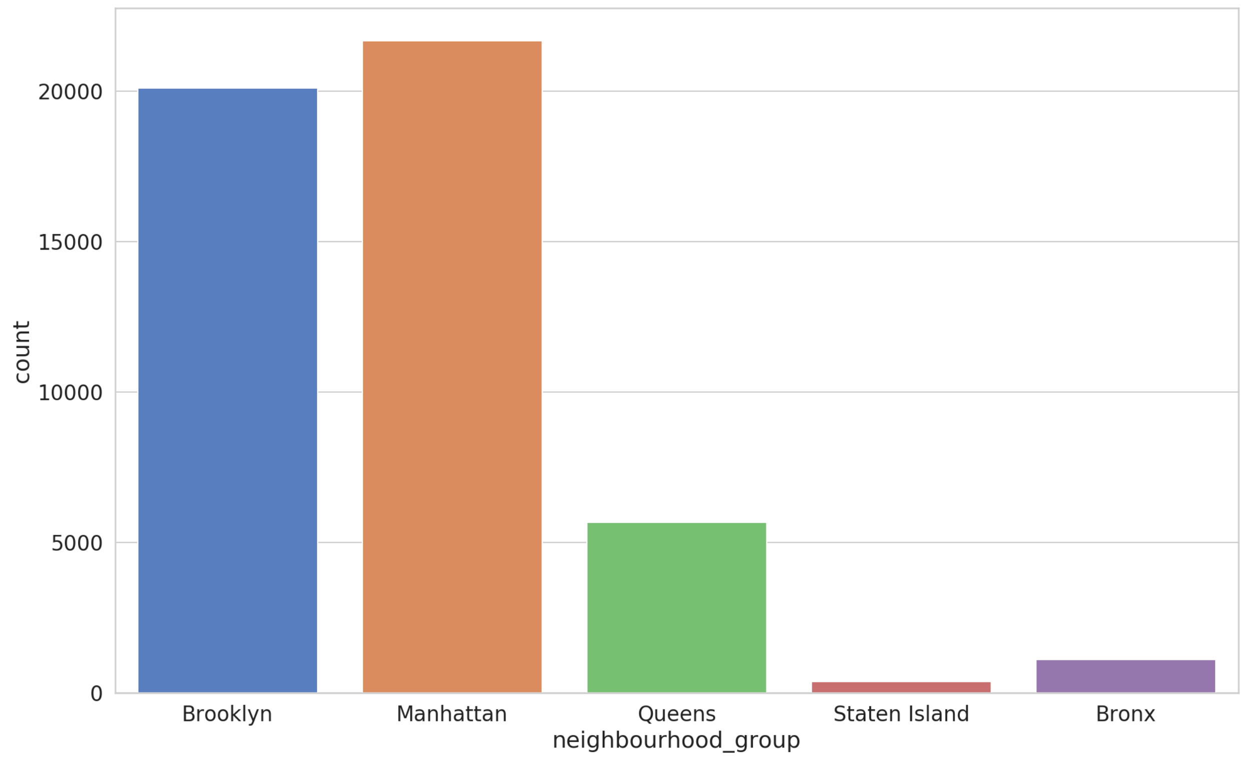
```python
sns.countplot(x='room_type', data=df)
```

[5]https://datascience.stackexchange.com/questions/40089/what-is-the-reason-behind-taking-log-transformation-of-few-continuous-variables



Most listings are offering entire places or private rooms. What about the location? What neighborhood groups are most represented?

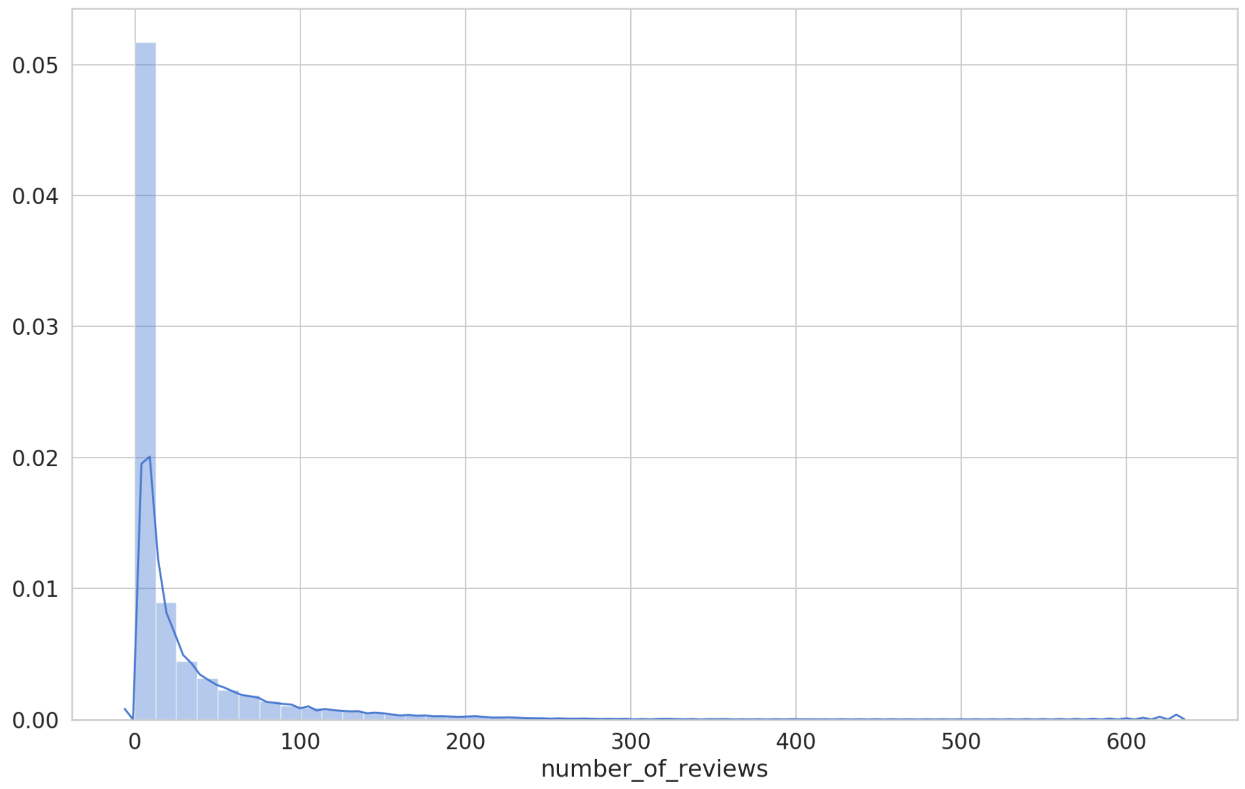
```
sns.countplot(x='neighbourhood_group', data=df)
```



As expected, Manhattan leads the way. Obviously, Brooklyn is very well represented, too. You can thank Mos Def, Nas, Masta Ace, and Fabolous for that.

Another interesting feature is the number of reviews. Let's have a look at it:

```
sns.distplot(df.number_of_reviews)
```



This one seems to follow a Power law[6] (it has a fat tail). This one seems to follow a Power law[7] (it has a fat tail). There seem to be some outliers (on the right) that might be of interest for investigation.

Finding Correlations

The correlation analysis might give you hints at what features might have predictive power when training your model.

> Remember, Correlation does not imply causation[8]

Computing Pearson correlation coefficient[9] between a pair of features is easy:

```
corr_matrix = df.corr()
```

Let's look at the correlation of the price with the other attributes:

[6]https://en.wikipedia.org/wiki/Power_law
[7]https://en.wikipedia.org/wiki/Power_law
[8]https://en.wikipedia.org/wiki/Correlation_does_not_imply_causation
[9]https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

```
price_corr = corr_matrix['price']
price_corr.iloc[price_corr.abs().argsort()]
```

```
latitude                          0.033939
minimum_nights                    0.042799
number_of_reviews                -0.047954
calculated_host_listings_count    0.057472
availability_365                  0.081829
longitude                        -0.150019
price                             1.000000
```

The correlation coefficient is defined in the -1 to 1 range. A value close to 0 means there is no correlation. Value of 1 suggests a perfect positive correlation (e.g. as the price of Bitcoin increases, your dreams of owning more are going up, too!). Value of -1 suggests perfect negative correlation (e.g. high number of bad reviews should correlate with lower prices).

The correlation in our dataset looks really bad. Luckily, categorical features are not included here. They might have some predictive power too! How can we use them?

Prepare the data

The goal here is to transform the data into a form that is suitable for your model. There are several things you want to do when handling (think CSV, Database) structured data:

- Handle missing data
- Remove unnecessary columns
- Transform any categorical features to numbers/vectors
- Scale numerical features

Missing data

Let's start with a check for missing data:

```
missing = df.isnull().sum()
missing[missing > 0].sort_values(ascending=False)
```

```
reviews_per_month    10052
last_review          10052
host_name               21
name                    16
```

We'll just go ahead and remove those features for this example. In real-world applications, you should consider other approaches.

```
df = df.drop([
    'id', 'name', 'host_id', 'host_name',
    'reviews_per_month', 'last_review', 'neighbourhood'
], axis=1)
```

We're also dropping the neighbourhood, host id (too many unique values), and the id of the listing.

Next, we're splitting the data into features we're going to use for the prediction and a target variable y (the price):

```
X = df.drop('price', axis=1)
y = np.log1p(df.price.values)
```

Note that we're applying the log transformation to the price.

Feature scaling and categorical data

Let's start with feature scaling[10]. Specifically, we'll do min-max normalization and scale the features in the 0-1 range. Luckily, the MinMaxScaler[11] from scikit-learn does just that.

But why do feature scaling at all? Largely because of the algorithm we're going to use to train our model[12] will do better with it.

Next, we need to preprocess the categorical data. Why?

Some Machine Learning algorithms can operate on categorical data without any preprocessing (like Decision trees, Naive Bayes). But most can't.

Unfortunately, you can't replace the category names with a number. Converting Brooklyn to 1 and Manhattan to 2 suggests that Manhattan is greater (2 times) than Brooklyn. That doesn't make sense. How can we solve this?

We can use One-hot encoding[13]. To get a feel of what it does, we'll use OneHotEncoder[14] from scikit-learn:

[10]https://en.wikipedia.org/wiki/Feature_scaling
[11]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
[12]https://arxiv.org/abs/1502.03167
[13]https://en.wikipedia.org/wiki/One-hot
[14]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

```
from sklearn.preprocessing import OneHotEncoder

data = [['Manhattan'], ['Brooklyn']]

OneHotEncoder(sparse=False).fit_transform(data)
```

```
array([[0., 1.],
       [1., 0.]])
```

Essentially, you get a vector for each value that contains 1 at the index of the category and 0 for every other value. This encoding solves the comparison issue. The negative part is that your data now might take much more memory.

All data preprocessing steps are to be performed on the training data and data we're going to receive via the REST API for prediction. We can unite the steps using `make_column_transformer()`[15]:

```
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
from sklearn.compose import make_column_transformer

transformer = make_column_transformer(
    (MinMaxScaler(), [
      'latitude', 'longitude', 'minimum_nights',
      'number_of_reviews', 'calculated_host_listings_count', 'availability_365'
    ]),
    (OneHotEncoder(handle_unknown="ignore"), [
      'neighbourhood_group', 'room_type'
    ])
)
```

We enumerate all columns that need feature scaling and one-hot encoding. Those columns will be replaced with the ones from the preprocessing steps. Next, we'll learn the ranges and categorical mapping using our transformer:

```
transformer.fit(X)
```

Finally, we'll transform our data:

```
transformer.transform(X)
```

The last thing is to separate the data into training and test sets:

[15]https://scikit-learn.org/stable/modules/generated/sklearn.compose.make_column_transformer.html

```
X_train, X_test, y_train, y_test =\
 train_test_split(X, y, test_size=0.2, random_state=RANDOM_SEED)
```

You're going to use only the training set while developing and evaluating your model. The test set will be used later.

That's it! You are now ready to build a model. How can you do that?

Build your model

Finally, it is time to do some modeling. Recall the goal we set for ourselves at the beginning:

> We're trying to predict Airbnb[16] listing price per night in NYC

We have a price prediction problem on our hands. More generally, we're trying to predict a numerical value defined in a very large range. This fits nicely in the Regression Analysis[17] framework.

Training a model boils down to minimizing some predefined error. What error should we measure?

Error measurement

We'll use Mean Squared Error[18] which measures the difference between average squared predicted and true values:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

where n is the number of samples, Y is a vector containing the real values and $\hat{Y}$ is a vector containing the predictions from our model.

Now that you have a measurement of how well your model is performing is time to build the model itself. How can you build a Deep Neural Network with Keras?

Build a Deep Neural Network with Keras

Keras[19] is the official high-level API for TensorFlow[20]. In short, it allows you to build complex models using a sweet interface. Let's build a model with it:

[16]https://www.airbnb.com/
[17]https://en.wikipedia.org/wiki/Regression_analysis
[18]https://en.wikipedia.org/wiki/Mean_squared_error
[19]https://keras.io/
[20]https://www.tensorflow.org/

```
model = keras.Sequential()
model.add(keras.layers.Dense(
  units=64,
  activation="relu",
  input_shape=[X_train.shape[1]]
))
model.add(keras.layers.Dropout(rate=0.3))
model.add(keras.layers.Dense(units=32, activation="relu"))
model.add(keras.layers.Dropout(rate=0.5))

model.add(keras.layers.Dense(1))
```

The sequential API allows you to add various layers to your model, easily. Note that we specify the *input_size* in the first layer using the training data. We also do regularization using Dropout layers[21].

How can we specify the error metric?

```
model.compile(
    optimizer=keras.optimizers.Adam(0.0001),
    loss = 'mae',
    metrics = ['mae'])
```

The `compile()`[22] method lets you specify the optimizer and the error metric you need to reduce.

Your model is ready for training. Let's go!

Training

Training a Keras model involves calling a single method - `fit()`[23]:

```
BATCH_SIZE = 32

early_stop = keras.callbacks.EarlyStopping(
  monitor='val_mae',
  mode="min",
  patience=10
)

history = model.fit(
```

[21]https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout
[22]https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile
[23]https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

```
  x=X_train,
  y=y_train,
  shuffle=True,
  epochs=100,
  validation_split=0.2,
  batch_size=BATCH_SIZE,
  callbacks=[early_stop]
)
```

We feed the training method with the training data and specify the following parameters:

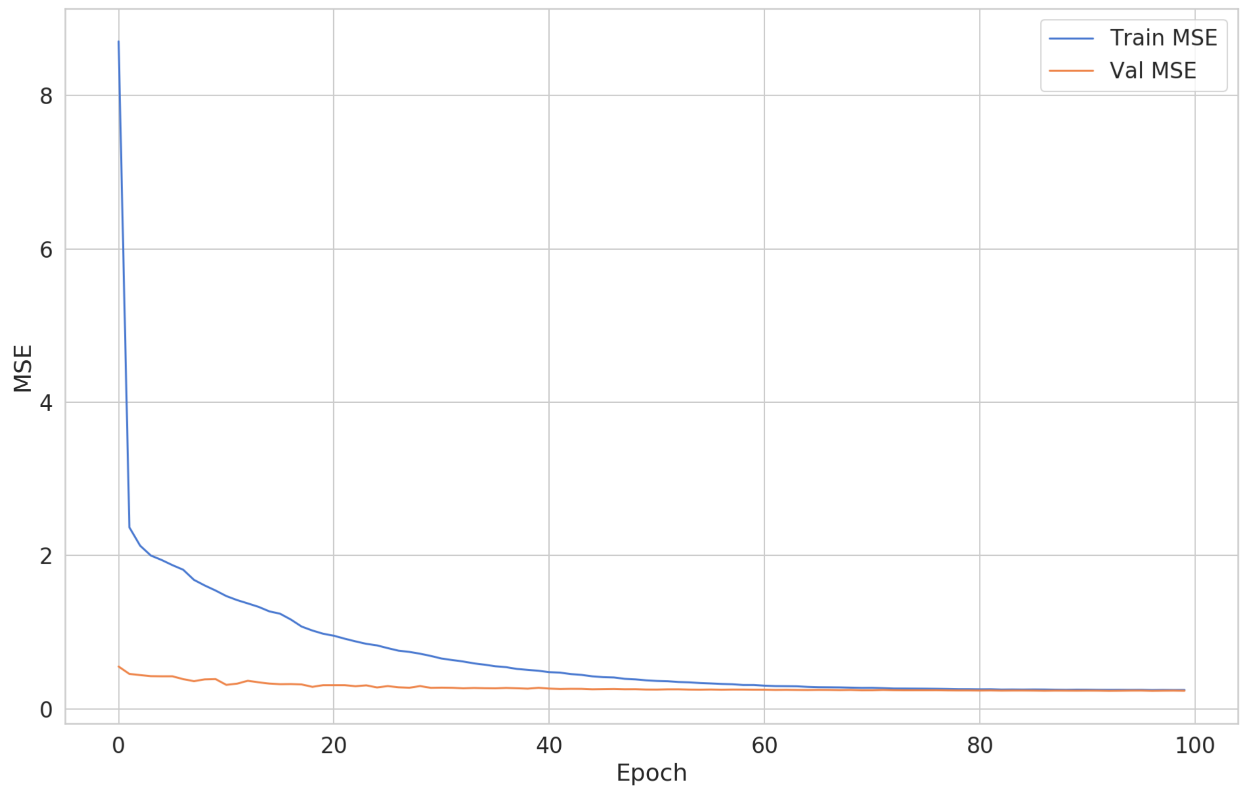
- shuffle - random sort the data
- epochs - number of training cycles
- validation_split - use some percent of the data for measuring the error and not during training
- batch_size - the number of training examples that are fed at a time to our model
- callbacks - we use EarlyStopping[24] to prevent our model from overfitting when the training and validation error start to diverge

After the long training process is complete, you need to answer one question. Can your model make good predictions?

Evaluation

One simple way to understand the training process is to look at the training and validation loss:

[24]https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping



We can see a large improvement in the training error, but not much on the validation error. What else can we use to test our model?

Using the test data

Recall that we have some additional data. Now it is time to use it and test how good our model. *Note that we don't use that data during the training, only once at the end of the process.*

Let's get the predictions from the model:

```python
y_pred = model.predict(X_test)
```

And we'll use a couple of metrics for the evaluation:

```python
from sklearn.metrics import mean_squared_error
from math import sqrt
from sklearn.metrics import r2_score

print(f'MSE {mean_squared_error(y_test, y_pred)}')
print(f'RMSE {np.sqrt(mean_squared_error(y_test, y_pred))}')
```

```
MSE 0.2139184014903989
RMSE 0.4625131365598159
```

We've already discussed MSE. You can probably guess what Root Mean Squared Error (RMSE)[25] means. RMSE allows us to penalize points further from the mean.

Another statistic we can use to measure how well our predictions fit with the real data is the R^2 score[26]. A value close to 1 indicates a perfect fit. Let's check ours:

```
print(f'R2 {r2_score(y_test, y_pred)}')
```

```
R2 0.5478250409482018
```

There is definitely room for improvement here. You might try to tune the model better and get better results.

Now you have a model and a rough idea of how well will it do in production. How can you save your work?

Save the model

Now that you have a trained model, you need to store it and be able to reuse it later. Recall that we have a data transformer that needs to be stored, too! Let's save both:

```
import joblib

joblib.dump(transformer, "data_transformer.joblib")
model.save("price_prediction_model.h5")
```

The recommended approach of storing scikit-learn models[27] is to use joblib[28]. Saving the model architecture and weights of a Keras model is done with the `save()`[29] method.

You can download the files from the notebook using the following:

[25]https://en.wikipedia.org/wiki/Root-mean-square_deviation
[26]https://en.wikipedia.org/wiki/Coefficient_of_determination
[27]https://scikit-learn.org/stable/modules/model_persistence.html#persistence-example
[28]https://joblib.readthedocs.io/en/latest/
[29]https://www.tensorflow.org/api_docs/python/tf/keras/Sequential#save

```python
from google.colab import files

files.download("data_transformer.joblib")
files.download("price_prediction_model.h5")
```

Build REST API

Building a REST API[30] allows you to use your model to make predictions for different clients. Almost any device can speak REST - Android, iOS, Web browsers, and many others.

Flask[31] allows you to build a REST API in just a couple of lines. *Of course, we're talking about a quick-and-dirty prototype.* Let's have a look at the complete code:

```python
from math import expm1

import joblib
import pandas as pd
from flask import Flask, jsonify, request
from tensorflow import keras

app = Flask(__name__)
model = keras.models.load_model("assets/price_prediction_model.h5")
transformer = joblib.load("assets/data_transformer.joblib")


@app.route("/", methods=["POST"])
def index():
    data = request.json
    df = pd.DataFrame(data, index=[0])
    prediction = model.predict(transformer.transform(df))
    predicted_price = expm1(prediction.flatten()[0])
    return jsonify({"price": str(predicted_price)})
```

The complete project (including the data transformer and model) is on GitHub: Deploy Keras Deep Learning Model with Flask[32]

The API has a single route (index) that accepts only POST requests. *Note that we pre-load the data transformer and the model.*

[30]https://en.wikipedia.org/wiki/Representational_state_transfer
[31]https://www.fullstackpython.com/flask.html
[32]https://github.com/curiousily/Deploy-Keras-Deep-Learning-Model-with-Flask

The request handler obtains the JSON data and converts it into a Pandas DataFrame. Next, we use the transformer to pre-process the data and get a prediction from our model. We invert the log operation we did in the pre-processing step and return the predicted price as JSON.

Your REST API is ready to go. Run the following command in the project directory:

```
flask run
```

Open a new tab to test the API:

```
curl -d '{"neighbourhood_group": "Brooklyn", "latitude": 40.64749, "longitude": -73.\
97237, "room_type": "Private room", "minimum_nights": 1, "number_of_reviews": 9, "ca\
lculated_host_listings_count": 6, "availability_365": 365}' -H "Content-Type: applic\
ation/json" -X POST http://localhost:5000
```

You should see something like the following:

```
{"price":"72.70381414559431"}
```

Great. How can you deploy your project and allow others to consume your model predictions?

Deploy to production

We'll deploy the project to Google App Engine[33]:

> App Engine enables developers to stay more productive and agile by supporting popular development languages and a wide range of developer tools.

App Engine allows us to use Python and easily deploy a Flask app.

You need to:

- Register for Google Cloud Engine account[34]
- Google Cloud SDK installed[35]

Here is the complete `app.yaml` config:

[33] https://cloud.google.com/appengine/
[34] https://cloud.google.com/compute/
[35] https://cloud.google.com/sdk/install

```
entrypoint: "gunicorn -b :$PORT app:app --timeout 500"
runtime: python
env: flex
service: nyc-price-prediction
runtime_config:
  python_version: 3.7
instance_class: B1
manual_scaling:
  instances: 1
liveness_check:
  path: "/liveness_check"
```

Execute the following command to deploy the project:

```
gcloud app deploy
```

Wait for the process to complete and test the API running on production. You did it!

Conclusion

Your model should now be running, making predictions, and accessible to everyone. Of course, you have a quick-and-dirty prototype. You will need a way to protect and monitor your API. Maybe you need a better (automated) deployment strategy too!

Still, you have a model deployed in production and did all of the following:

- Define your goal
- Load data
- Data exploration
- Data preparation
- Build and evalute your model
- Save the model
- Build REST API
- Deploy to production

How do you deploy your models? Comment down below :)

Run the modeling code in your browser[36]

The complete project on GitHub[37]

[36]https://colab.research.google.com/drive/1YxCmQb2YKh7VuQ_XgPXhEeIM3LpjV-mS
[37]https://github.com/curiousily/Deploy-Keras-Deep-Learning-Model-with-Flask

References

- Joblib - running Python functions as pipeline jobs[38]
- Flask - lightweight web application framework[39]
- Building a simple Keras + deep learning REST API[40]

[38]https://joblib.readthedocs.io/en/latest/
[39]https://palletsprojects.com/p/flask/
[40]https://blog.keras.io/building-a-simple-keras-deep-learning-rest-api.html

Object Detection

> TL;DR Learn how to prepare a custom dataset for object detection and detect vehicle plates. Use transfer learning to finetune the model and make predictions on test images.

Detecting objects in images and video is a hot research topic and really useful in practice. The advancement in Computer Vision (CV) and Deep Learning (DL) made training and running object detectors possible for practitioners of all scale. Modern object detectors are both fast and much more accurate (actually, usefully accurate).

This guide shows you how to fine-tune a pre-trained Neural Network on a large Object Detection dataset. We'll learn how to detect vehicle plates from raw pixels. Spoiler alert, the results are not bad at all!

You'll learn how to prepare a custom dataset and use a library for object detection based on TensorFlow and Keras. Along the way, we'll have a deeper look at what Object Detection is and what models are used for it.

Here's what will do:

- Understand Object Detection
- RetinaNet
- Prepare the Dataset
- Train a Model to Detect Vehicle Plates

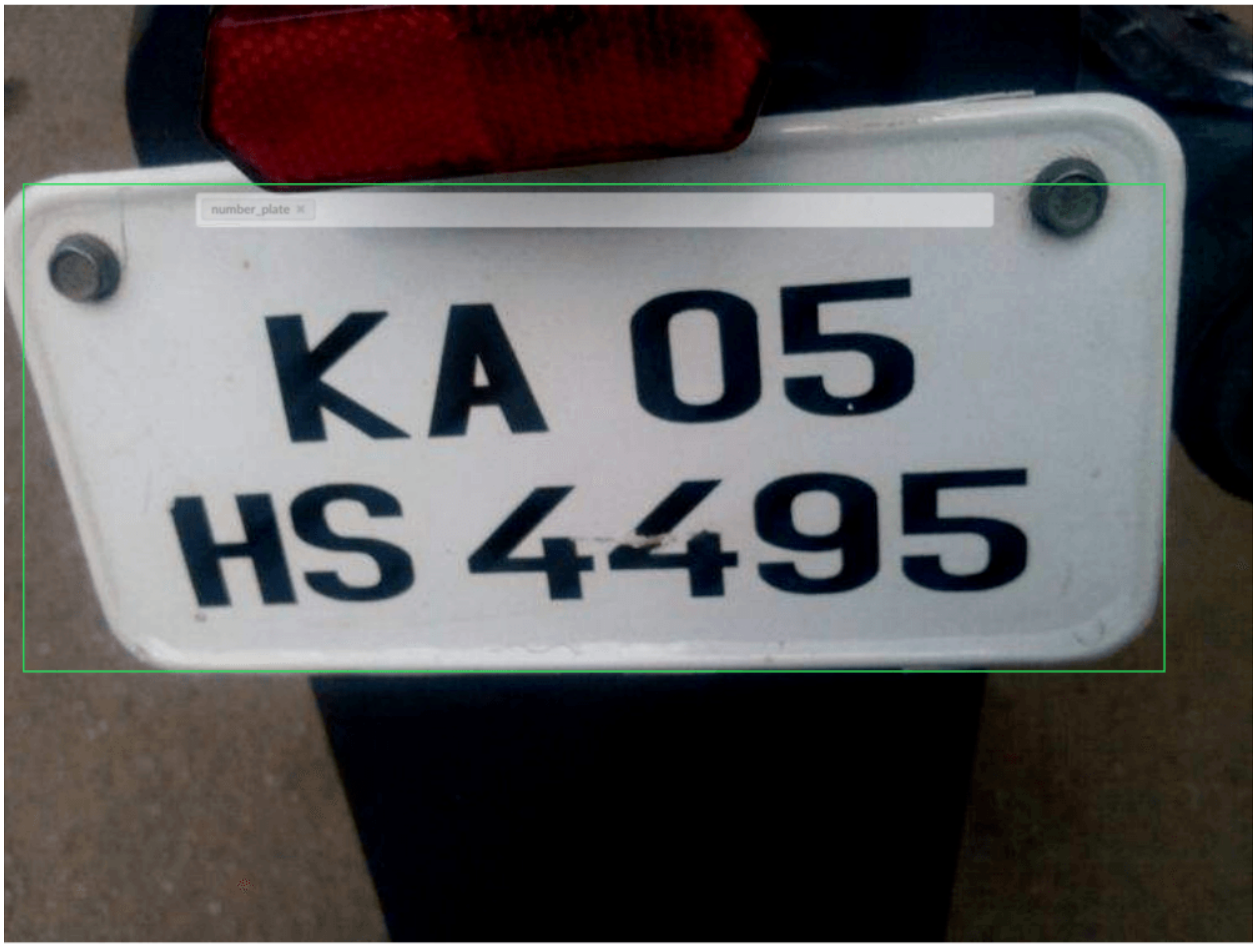
Run the complete notebook in your browser[41]

The complete project on GitHub[42]

Object Detection

Object detection[43] methods try to find the best bounding boxes around objects in images and videos. It has a wide array of practical applications - face recognition, surveillance, tracking objects, and more.

[41]https://colab.research.google.com/drive/1ldnii3sGJaUHPV6TWImykbeE_O-8VIlN
[42]https://github.com/curiousily/Deep-Learning-For-Hackers
[43]https://en.wikipedia.org/wiki/Object_detection



A lot of classical approaches have tried to find fast and accurate solutions to the problem. Sliding windows for object localization and image pyramids for detection at different scales are one of the most used ones. Those methods were slow, error-prone, and not able to handle object scales very well.

Deep Learning changed the field so much that it is now relatively easy for the practitioner to train models on small-ish datasets and achieve high accuracy and speed.

Usually, the result of object detection contains three elements:

- list of bounding boxes with coordinates
- the category/label for each bounding box
- the confidence score (0 to 1) for each bounding box and label

How can you evaluate the performance of object detection models?

Evaluating Object Detection

The most common measurement you'll come around when looking at object detection performance is Intersection over Union (IoU). This metric can be evaluated independently of the algorithm/model used.

The IoU is a ratio given by the following equation:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

IoU allows you to evaluate how well two bounding boxes overlap. In practice, you would use the annotated (true) bounding box, and the detected/predicted one. A value close to 1 indicates a very good overlap while getting closer to 0 gives you almost no overlap.

Getting *IoU* of 1 is very unlikely in practice, so don't be too harsh on your model.

Mean Average Precision (mAP)

Reading papers and leaderboards on Object Detection will inevitably lead you to an *mAP* value report. Typically, you'll see something like **mAP@0.5** indicating that object detection is considered correct only when this value is greater than 0.5.

The value is derived by averaging the precision of each class in the dataset. We can get the average precision for a single class by computing the *IoU* for every example in the class and divide by the number of class examples. Finally, we can get *mAP* by dividing by the number of classes.

RetinaNet

RetinaNet, presented by Facebook AI Research in Focal Loss for Dense Object Detection (2017)[44], is an object detector architecture that became very popular and widely used in practice. Why is RetinaNet so special?

RetinaNet is a one-stage detector. The most successful object detectors up to this point were operating on two stages (R-CNNs). The first stage involves selecting a set of regions (candidates) that might contain objects of interest. The second stage applies a classifier to the proposals.

One stage detectors (like RetinaNet) skip the region selection steps and runs detection over a lot of possible locations. This is faster and simpler but might reduce the overall prediction performance of the model.

RetinaNet is built on top of two crucial concepts - *Focal Loss* and *Featurized Image Pyramid*:

- **Focal Loss** is designed to mitigate the issue of extreme imbalance between background and foreground with objects of interest. It assigns more weight on hard, easily misclassified examples and small weight to easier ones.
- The **Featurized Image Pyramid** is the vision component of RetinaNet. It allows for object detection at different scales by stacking multiple convolutional layers.

[44]https://arxiv.org/pdf/1708.02002v2.pdf

Keras Implementation

Let's get real. RetinaNet is not a SOTA model for object detection. Not by a long shot[45]. However, well maintained, bug-free, and easy to use implementation of a good-enough model can give you a good estimate of how well you can solve your problem. In practice, you want a good-enough solution to your problem, and you (or your manager) wants it yesterday.

Keras RetinaNet[46] is a well maintained and documented implementation of RetinaNet. Go and have a look at the Readme to get a feel of what is capable of. It comes with a lot of pre-trained models and an easy way to train on custom datasets.

Preparing the Dataset

The task we're going to work on is vehicle number plate detection from raw images. Our data is hosted on Kaggle[47] and contains an annotation file with links to the images. Here's a sample annotation:

```
{
  "content": "http://com.dataturks.a96-i23.open.s3.amazonaws.com/2c9fafb0646e9cf9016\
473f1a561002a/77d1f81a-bee6-487c-aff2-0efa31a9925c____bd7f7862-d727-11e7-ad30-e18a56\
154311.jpg",
  "annotation": [
    {
      "label": [
        "number_plate"
      ],
      "notes": null,
      "points": [
        {
          "x": 0.7220843672456576,
          "y": 0.5879828326180258
        },
        {
          "x": 0.8684863523573201,
          "y": 0.6888412017167382
        }
      ],
      "imageWidth": 806,
      "imageHeight": 466
    }
```

[45]https://paperswithcode.com/sota/object-detection-on-coco
[46]https://github.com/fizyr/keras-retinanet
[47]https://www.kaggle.com/dataturks/vehicle-number-plate-detection

```
  ],
  "extras": null
}
```

This will require some processing to turn those xs and ys into proper image positions. Let's start with downloading the JSON file:

```
!gdown --id 1mTtB8GTWs74Yeqm0KMExGJZh1eDbzUlT --output indian_number_plates.json
```

We can use Pandas to read the JSON into a DataFrame:

```
plates_df = pd.read_json('indian_number_plates.json', lines=True)
```

Next, we'll download the images in a directory and create an annotation file for our training data in the format (expected by Keras RetinaNet):

```
path/to/image.jpg,x1,y1,x2,y2,class_name
```

Let's start by creating the directory:

```
os.makedirs("number_plates", exist_ok=True)
```

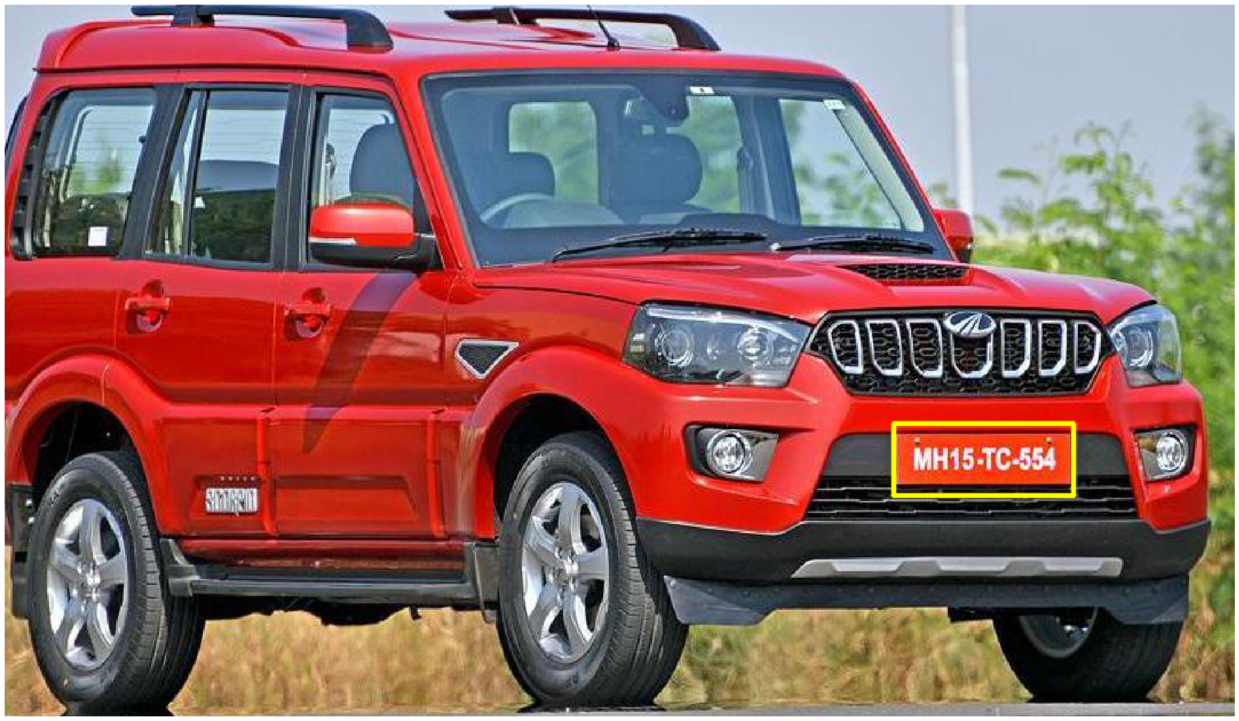
We can unify the download and the creation of annotation file like so:

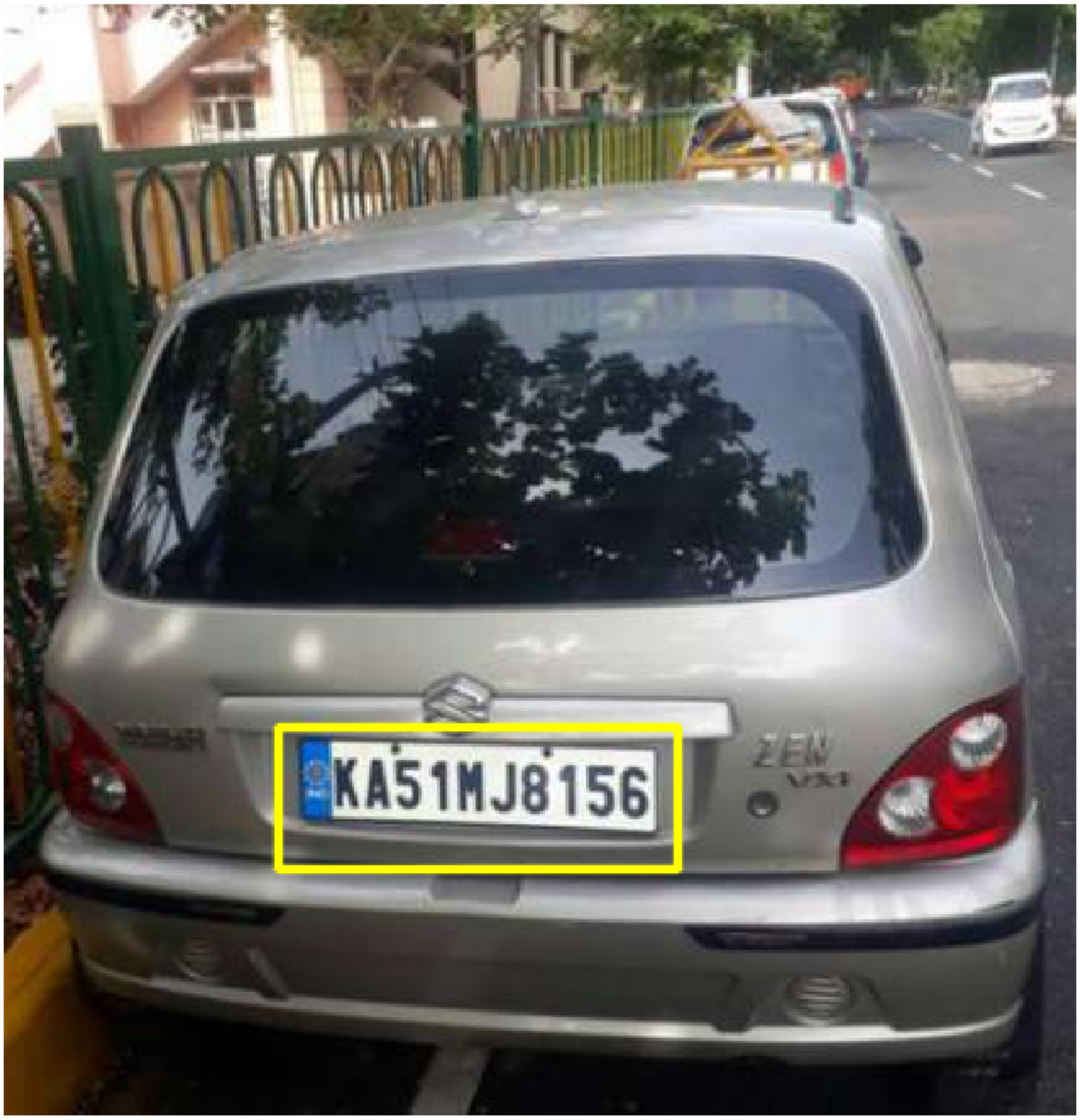
```
dataset = dict()
dataset["image_name"] = list()
dataset["top_x"] = list()
dataset["top_y"] = list()
dataset["bottom_x"] = list()
dataset["bottom_y"] = list()
dataset["class_name"] = list()

counter = 0
for index, row in plates_df.iterrows():
    img = urllib.request.urlopen(row["content"])
    img = Image.open(img)
    img = img.convert('RGB')
    img.save(f'number_plates/licensed_car_{counter}.jpeg', "JPEG")

    dataset["image_name"].append(
      f'number_plates/licensed_car_{counter}.jpeg'
    )

    data = row["annotation"]

    width = data[0]["imageWidth"]
    height = data[0]["imageHeight"]

    dataset["top_x"].append(
      int(round(data[0]["points"][0]["x"] * width))
    )
    dataset["top_y"].append(
      int(round(data[0]["points"][0]["y"] * height))
    )
    dataset["bottom_x"].append(
      int(round(data[0]["points"][1]["x"] * width))
    )
    dataset["bottom_y"].append(
      int(round(data[0]["points"][1]["y"] * height))
    )
    dataset["class_name"].append("license_plate")

    counter += 1
print("Downloaded {} car images.".format(counter))
```

We can use the dict to create a Pandas DataFrame:

```
df = pd.DataFrame(dataset)
```

Let's get a look at some images of vehicle plates:


MH15-TC-554



Preprocessing

We've already done a fair bit of preprocessing. A bit more is needed to convert the data into the format that Keras Retina understands:

```
path/to/image.jpg,x1,y1,x2,y2,class_name
```

First, let's split the data into training and test datasets:

```python
train_df, test_df = train_test_split(
  df,
  test_size=0.2,
  random_state=RANDOM_SEED
)
```

We need to write/create two CSV files for the annotations and classes:

```python
ANNOTATIONS_FILE = 'annotations.csv'
CLASSES_FILE = 'classes.csv'
```

We'll use Pandas to write the annotations file, excluding the index and header:

```python
train_df.to_csv(ANNOTATIONS_FILE, index=False, header=None)
```

We'll use regular old file writer for the classes:

```python
classes = set(['license_plate'])

with open(CLASSES_FILE, 'w') as f:
  for i, line in enumerate(sorted(classes)):
    f.write('{},{}\n'.format(line,i))
```

Detecting Vehicle Plates

You're ready to finetune the model on the dataset. Let's create a folder where we're going to store the model checkpoints:

```python
os.makedirs("snapshots", exist_ok=True)
```

You have two options at this point. Download the pre-trained model:

```
!gdown --id 1wPgOBoSks6bTIs9RzNvZf6HWROkciS8R --output snapshots/resnet50_csv_10.h5
```

Or train the model on your own:

```
PRETRAINED_MODEL = './snapshots/_pretrained_model.h5'

URL_MODEL = 'https://github.com/fizyr/keras-retinanet/releases/download/0.5.1/resnet\
50_coco_best_v2.1.0.h5'
urllib.request.urlretrieve(URL_MODEL, PRETRAINED_MODEL)

print('Downloaded pretrained model to ' + PRETRAINED_MODEL)
```

Here, we save the weights of the pre-trained model on the Coco[48] dataset.

The training script requires paths to the annotation, classes files, and the downloaded weights (along with other options):

```
!keras_retinanet/bin/train.py \
 --freeze-backbone \
 --random-transform \
 --weights {PRETRAINED_MODEL} \
 --batch-size 8 \
 --steps 500 \
 --epochs 10 \
 csv annotations.csv classes.csv
```

Make sure to choose an appropriate batch size, depending on your GPU. Also, the training might take a lot of time. Go get a hot cup of rakia, while waiting.

Loading the model

You should have a directory with some snapshots at this point. Let's take the most recent one and convert it into a format that Keras RetinaNet understands:

```
model_path = os.path.join(
  'snapshots',
  sorted(os.listdir('snapshots'), reverse=True)[0]
)

model = models.load_model(model_path, backbone_name='resnet50')
model = models.convert_model(model)
```

Your object detector is almost ready. The final step is to convert the classes into a format that will be useful later:

[48]http://cocodataset.org/

```
labels_to_names = pd.read_csv(
  CLASSES_FILE,
  header=None
).T.loc[0].to_dict()
```

Detecting objects

How good is your trained model? Let's find out by drawing some detected boxes along with the true/annotated ones. The first step is to get predictions from our model:

```
def predict(image):
  image = preprocess_image(image.copy())
  image, scale = resize_image(image)

  boxes, scores, labels = model.predict_on_batch(
    np.expand_dims(image, axis=0)
  )

  boxes /= scale

  return boxes, scores, labels
```

We're resizing and preprocessing the image using the tools provided by the library. Next, we need to add an additional dimension to the image tensor, since the model works on multiple/batch of images. We rescale the detected boxes based on the resized image scale. The function returns all predictions.

The next helper function will draw the detected boxes on top of the vehicle image:

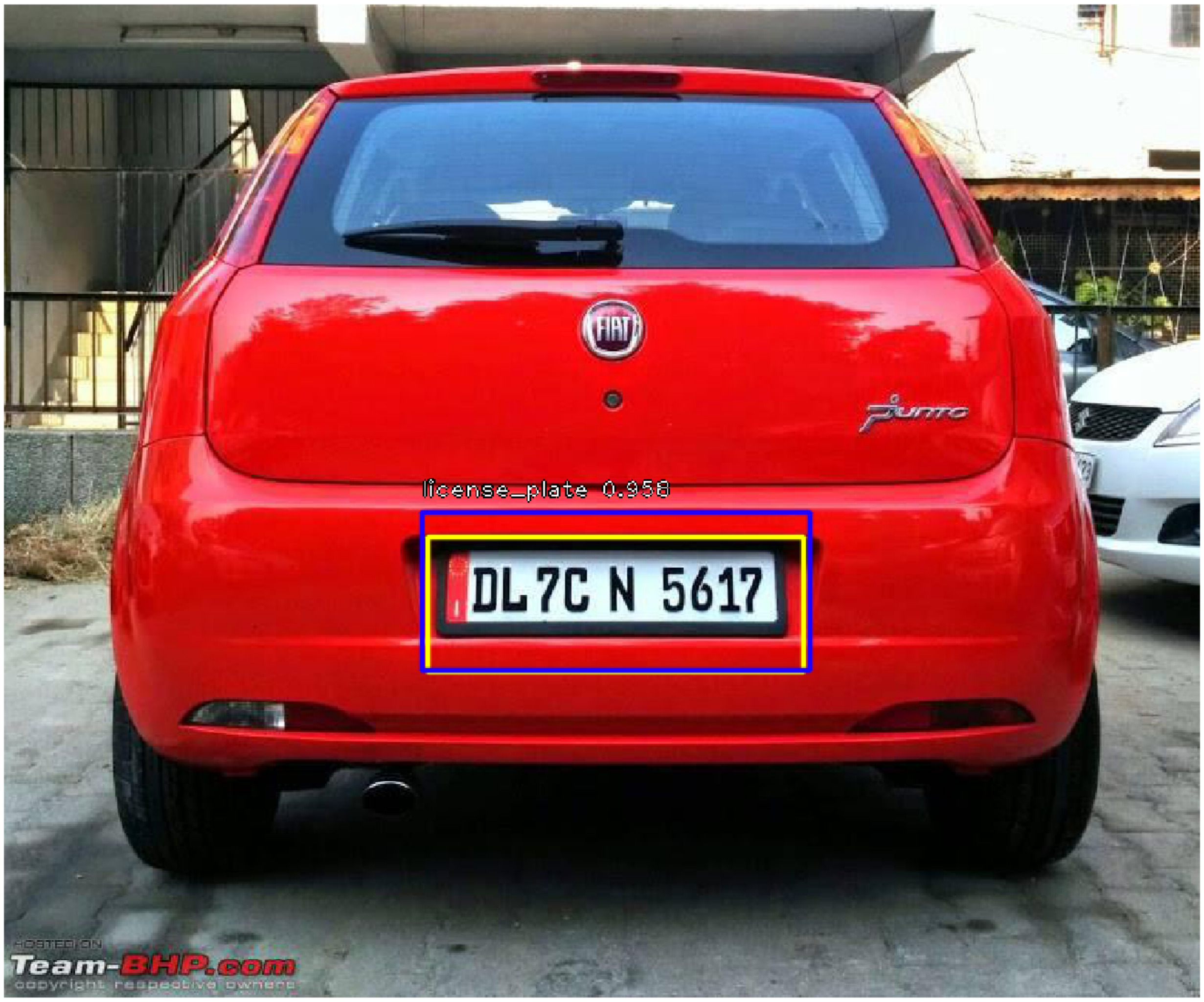
```
THRES_SCORE = 0.6

def draw_detections(image, boxes, scores, labels):
  for box, score, label in zip(boxes[0], scores[0], labels[0]):
    if score < THRES_SCORE:
        break

    color = label_color(label)

    b = box.astype(int)
    draw_box(image, b, color=color)

    caption = "{} {:.3f}".format(labels_to_names[label], score)
    draw_caption(image, b, caption)
```

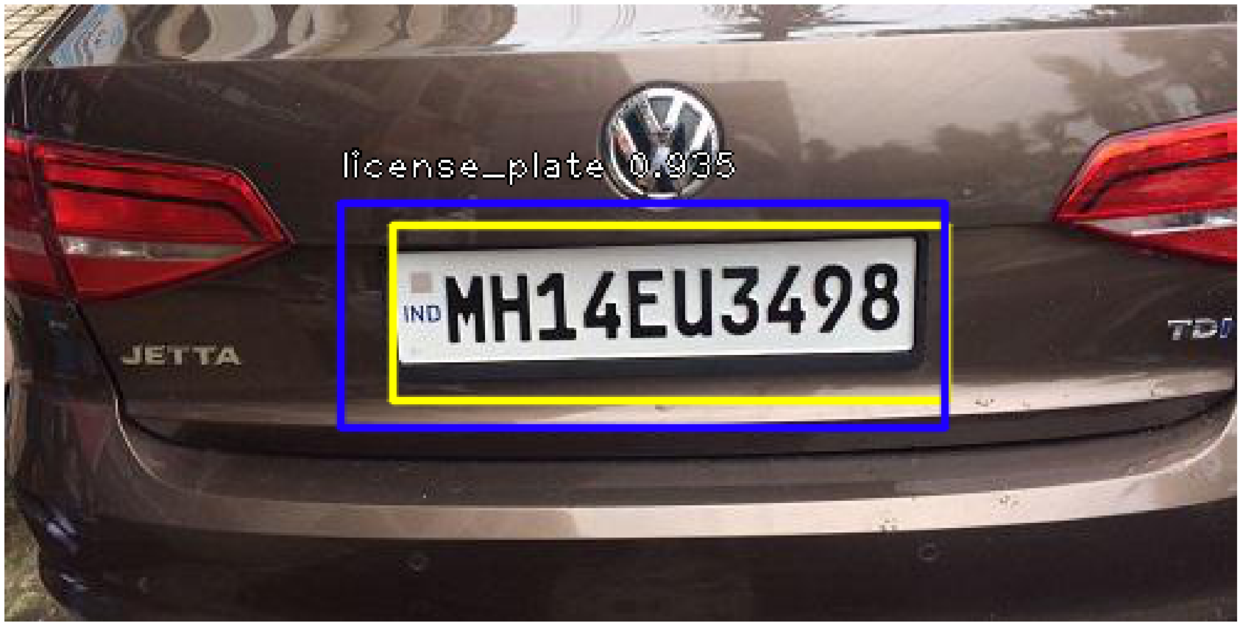
We'll draw detections with a confidence score above 0.6. Note that the scores are sorted high to low, so breaking from the loop is fine.

Let's put everything together:

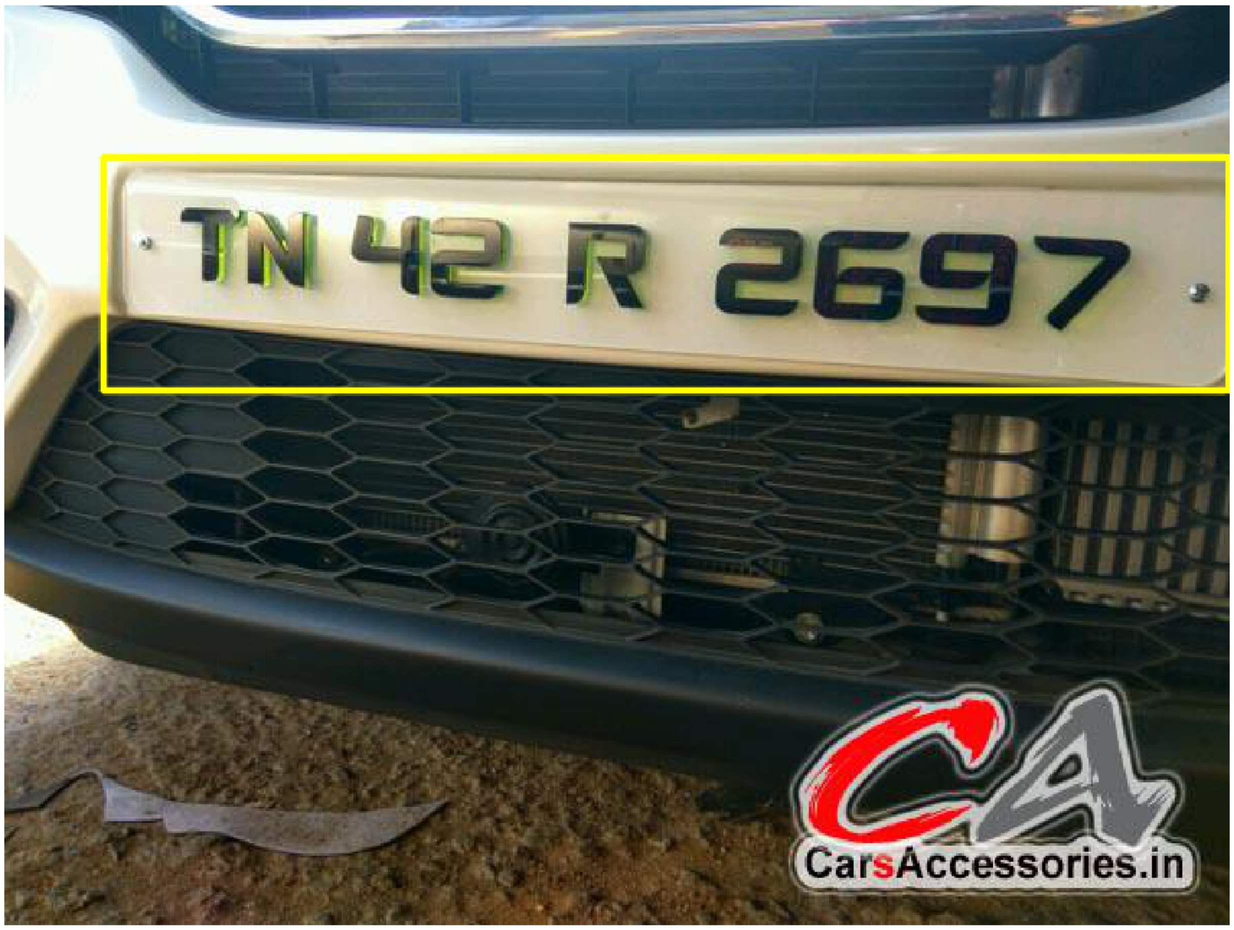
```python
def show_detected_objects(image_row):
  img_path = image_row.image_name

  image = read_image_bgr(img_path)

  boxes, scores, labels = predict(image)

  draw = image.copy()
  draw = cv2.cvtColor(draw, cv2.COLOR_BGR2RGB)

  true_box = [
    image_row.x_min, image_row.y_min, image_row.x_max, image_row.y_max
  ]
  draw_box(draw, true_box, color=(255, 255, 0))

  draw_detections(draw, boxes, scores, labels)

  plt.axis('off')
  plt.imshow(draw)
  plt.show()
```

Here are the results of calling this function on two examples from the test set:


license_plate 0.958
DL 7C N 5617
Team-BHP.com
copyright respective owners



Things look pretty good. Our detected boxes are colored in blue, while the annotations are in yellow. Before jumping to conclusions, let’s have a look at another example:



Our model didn't detect the plate on this vehicle. Maybe it wasn't confident enough? You can try to run the detection with a lower threshold.

Conclusion

Well done! You've built an Object Detector that can (somewhat) find vehicle number plates in images. You used a pre-trained model and fine tuned it on a small dataset to adapt it to the task at hand.

Here's what you did:

- Understand Object Detection
- RetinaNet
- Prepare the Dataset
- Train a Model to Detect Vehicle Plates

Can you use the concepts you learned here and apply it to a problem/dataset you have?

Run the complete notebook in your browser[49]

The complete project on GitHub[50]

References

- Keras RetinaNet[51]
- Vehicle Number Plate Detection[52]
- Object detection: speed and accuracy comparison[53]
- Focal Loss for Dense Object Detection[54]
- Plate Detection –> Preparing the data[55]
- Object Detection in Colab with Fizyr Retinanet[56]

[49]https://colab.research.google.com/drive/1ldnii3sGJaUHPV6TWImykbeE_O-8VIIN
[50]https://github.com/curiousily/Deep-Learning-For-Hackers
[51]https://github.com/fizyr/keras-retinanet
[52]https://www.kaggle.com/dataturks/vehicle-number-plate-detection
[53]https://medium.com/@jonathan_hui/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359
[54]https://arxiv.org/abs/1708.02002
[55]https://www.kaggle.com/dsousa/plate-detection-preparing-the-data
[56]https://www.freecodecamp.org/news/object-detection-in-colab-with-fizyr-retinanet-efed36ac4af3/