Venelin Valkov

Hacker's Guide to Machine Learning with Python

Contents

TensorFlow 2 and Keras - Quick Start Guide

> TL;DR Learn how to use Tensors, build a Linear Regression model and a simple Neural Network

TensorFlow 2.0 (final) was released at the end of September. Oh boy, it looks much cooler than the 1.x series. Why is it so much better for you, the developer?

- One high-level API for building models (that you know and love) - Keras. The good news is that most of your old Keras code should work automagically after changing a couple of imports.
- Eager execution - all your code looks much more like normal Python programs. Old-timers might remember the horrible `Session` experiences. You shouldn't need any of that, in day-to-day use.

There are tons of other improvements, but the new developer experience is something that will make using TensorFlow 2 sweeter. What about PyTorch? PyTorch is still great and easy to use. But it seems like TensorFlow is catching up, or is it?

You'll learn:

- How to install TensorFlow 2
- What is a Tensor
- Doing Tensor math
- Using probability distributions and sampling
- Build a Simple Linear Regression model
- Build a Simple Neural Network model
- Save/restore a model

Run the complete code in your browser[1]

Setup

Let's install the GPU-supported version and set up the environment:

[1] https://colab.research.google.com/drive/1HkG7HYS1-IFAYbECZ0zleBWA3Xi4DKIm

```
!pip install tensorflow-gpu
```

Check the installed version:

```
import tensorflow as tf

tf.__version__
```

```
2.0.0
```

And specify a random seed, so our results are reproducible:

```
RANDOM_SEED = 42

tf.random.set_seed(RANDOM_SEED)
```

Tensors

TensorFlow allows you to define and run operations on Tensors. Tensors are data-containers that can be of arbitrary dimension - scalars, vectors, matrices, etc. You can put numbers (floats and ints) and strings into Tensors.

Let's create a simple Tensor:

```
x = tf.constant(1)
print(x)
```

```
tf.Tensor(1, shape=(), dtype=int32)
```

It seems like our first Tensor contains the number 1, it is of type int32 and is shapeless. To obtain the value we can do:

```
x.numpy()
```

```
1
```

Let's create a simple matrix:

```
m = tf.constant([[1, 2, 1], [3, 4, 2]])
print(m)
```

```
tf.Tensor(
[[1 2 1]
 [3 4 2]], shape=(2, 3), dtype=int32)
```

This shape thingy seems to specify *rows x columns*. In general, the shape array shows how many elements are in every dimension of the Tensor.

Helpers

TensorFlow offers a variety of helper functions for creating Tensors. Let's create a matrix full of ones:

```
ones = tf.ones([3, 3])
print(ones)
```

```
tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]], shape=(3, 3), dtype=float32)
```

and zeros:

```
zeros = tf.zeros([2, 3])
print(zeros)
```

```
tf.Tensor(
[[0. 0. 0.]
 [0. 0. 0.]], shape=(2, 3), dtype=float32)
```

We have two rows and three columns. What if we want to turn it into three rows and two columns:

```
tf.reshape(zeros, [3, 2])
tf.Tensor(
[[0. 0.]
 [0. 0.]
 [0. 0.]], shape=(3, 2), dtype=float32)
```

You can use another helper function to replace rows and columns (transpose):

```
tf.transpose(zeros)
```

```
tf.Tensor(
[[0. 0.]
 [0. 0.]
 [0. 0.]], shape=(3, 2), dtype=float32)
```

Tensor Math

Naturally, you would want to do something with your data. Let's start with adding numbers:

```
a = tf.constant(1)
b = tf.constant(1)

tf.add(a, b).numpy()
```

```
42
```

That seems reasonable :) You can do the same thing using something more human friendly:

```
(a + b).numpy()
```

You can multiply Tensors like so:

```
c = a + b
c * c
```

And compute dot product of matrices:

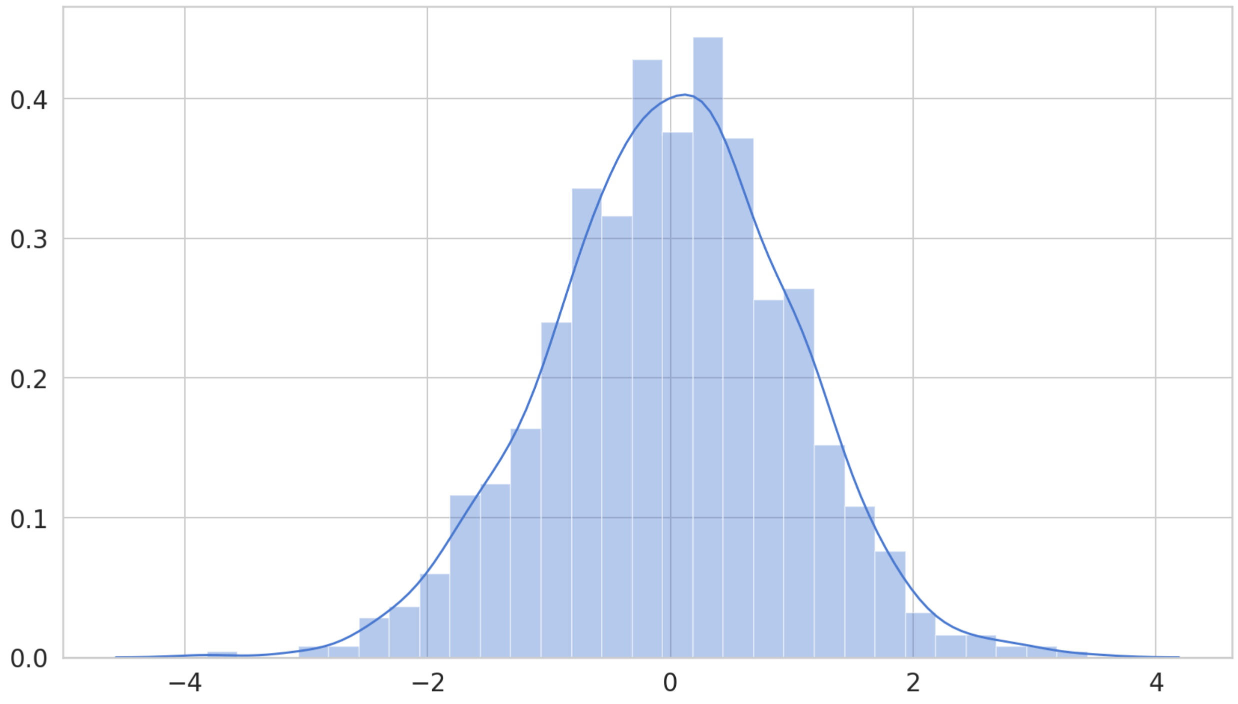
```
d1 = tf.constant([[1, 2], [1, 2]]);
d2 = tf.constant([[3, 4], [3, 4]]);

tf.tensordot(d1, d2, axes=1).numpy()
```

```
array([[ 9, 12],
       [ 9, 12]], dtype=int32)
```

Sampling

You can also generate random numbers according to some famous probability distributions. Let's start with Normal[2]:

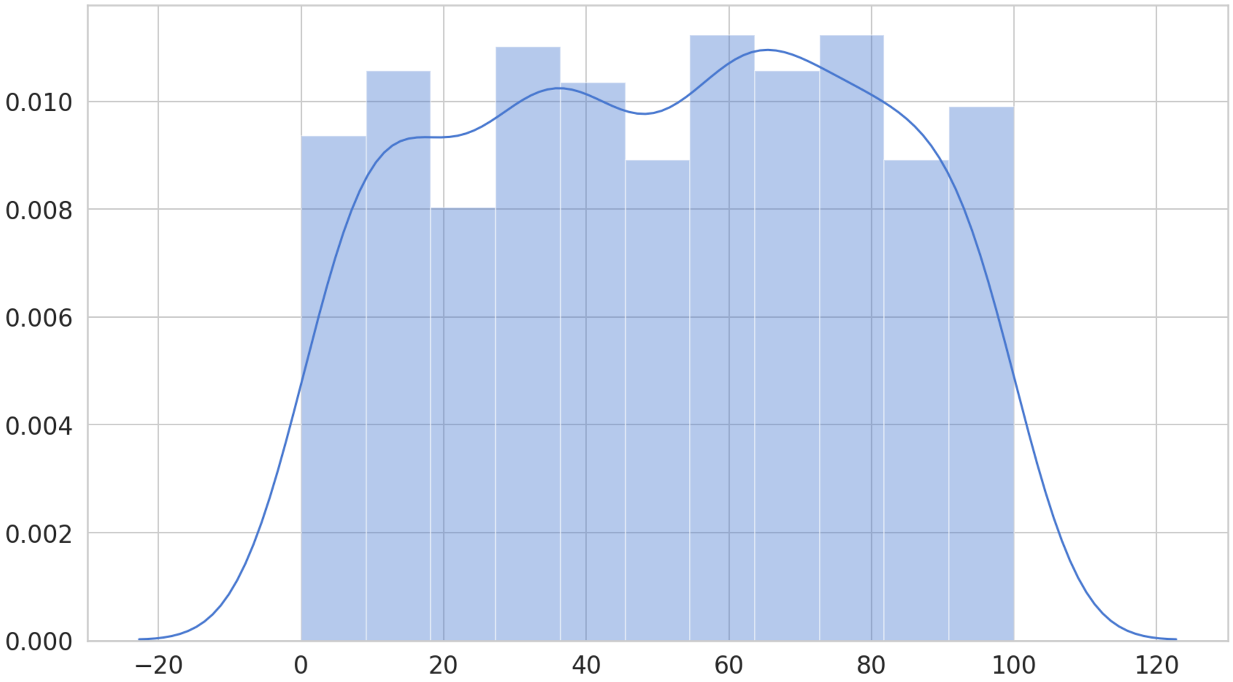
```
norm = tf.random.normal(shape=(1000, 1), mean=0., stddev=1.)
```



We can do the same thing from the Uniform[3]:

[2] https://en.wikipedia.org/wiki/Normal_distribution
[3] https://en.wikipedia.org/wiki/Uniform_distribution_(continuous)

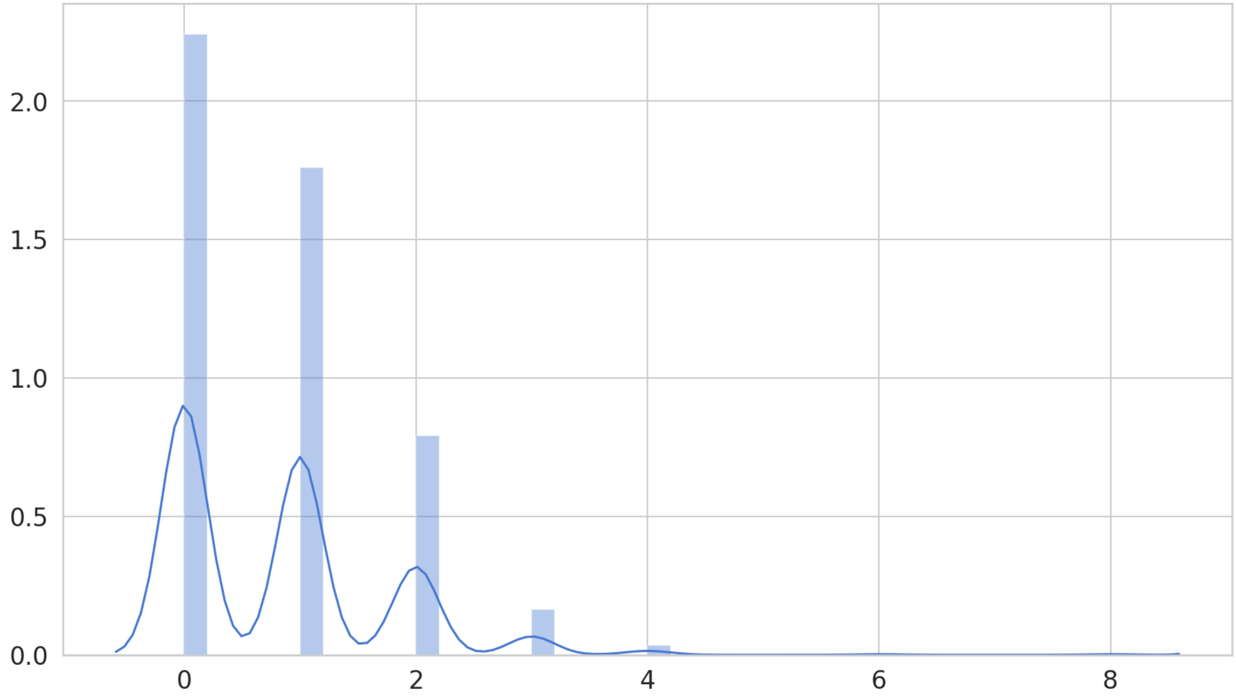
```
unif = tf.random.uniform(shape=(1000, 1), minval=0, maxval=100)
```



Let's have a look at something a tad more exotic - the Poisson distribution[4]. It is popular for modeling the number of times an event occurs in some time. It is the first one (in our exploration) that contains a hyperparameter - λ. It controls the number of expected occurrences.

```
pois = tf.random.poisson(shape=(1000, 1), lam=0.8)
```

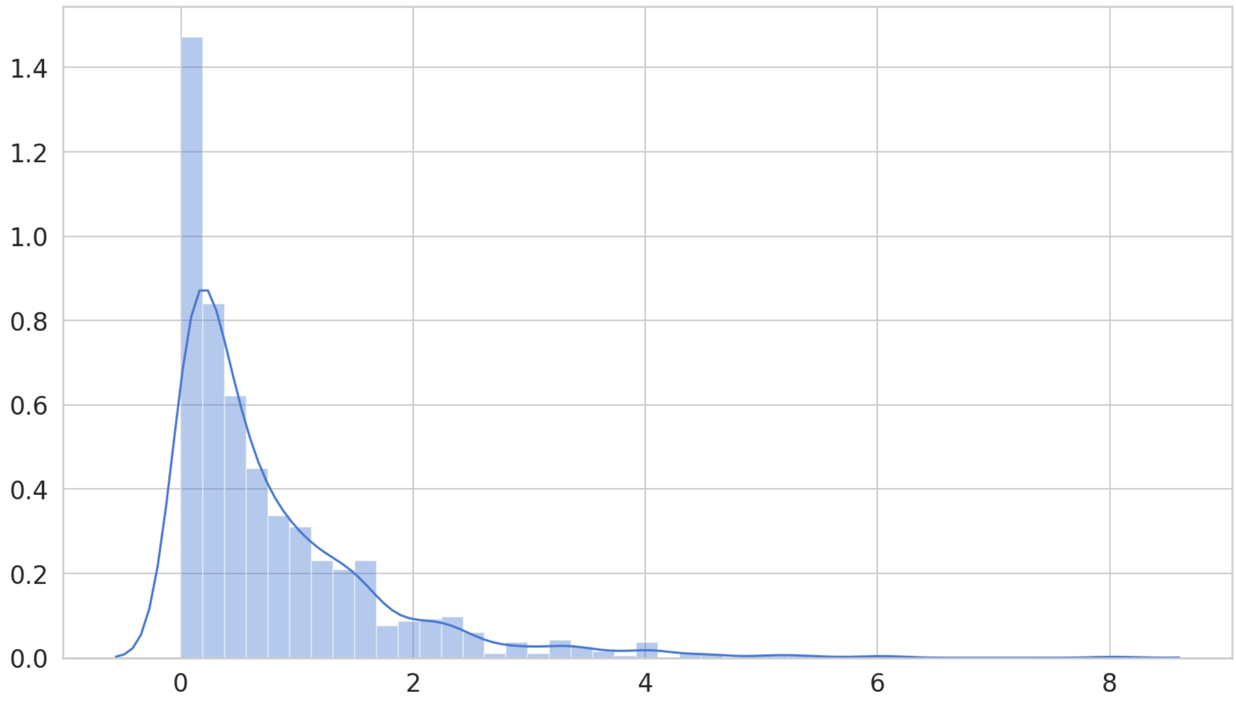
[4] https://en.wikipedia.org/wiki/Poisson_distribution



The Gamma distribution[5] is continuous. It has 2 hyperparameters that control the shape and scale. It is used to model always positive continuous variables with skewed distributions.

```
gam = tf.random.gamma(shape=(1000, 1), alpha=0.8)
```

[5] https://en.wikipedia.org/wiki/Gamma_distribution



Simple Linear Regression Model

Let's build a Simple Linear Regression[6] model to predict the stopping distance of cars based on their speed. The data comes from here: https://vincentarelbundock.github.io/Rdatasets/datasets.html[7]. It is given by this Tensor:

```
data = tf.constant([
  [4,2],
  [4,10],
  [7,4],
  [7,22],
  [8,16],
  [9,10],
  [10,18],
  [10,26],
  [10,34],
  [11,17],
  [11,28],
  [12,14],
  [12,20],
```

[6] https://en.wikipedia.org/wiki/Simple_linear_regression

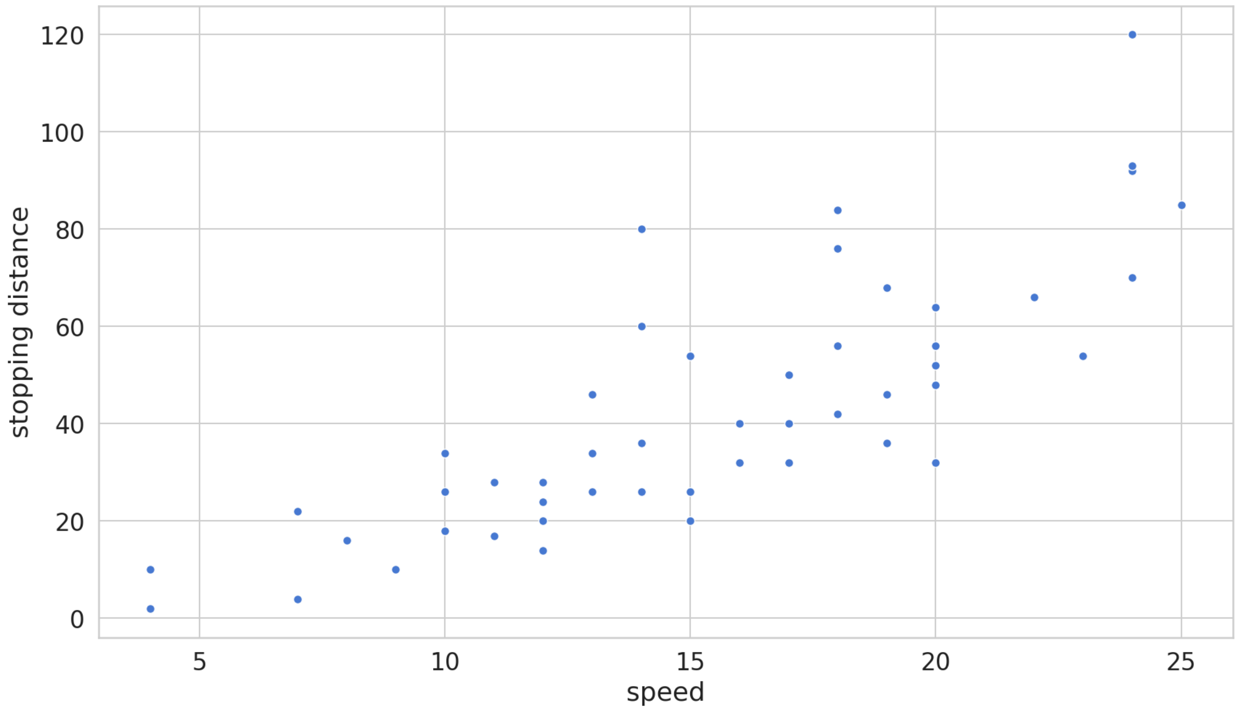
[7] https://vincentarelbundock.github.io/Rdatasets/datasets.html

```
  [12,24],
  [12,28],
  [13,26],
  [13,34],
  [13,34],
  [13,46],
  [14,26],
  [14,36],
  [14,60],
  [14,80],
  [15,20],
  [15,26],
  [15,54],
  [16,32],
  [16,40],
  [17,32],
  [17,40],
  [17,50],
  [18,42],
  [18,56],
  [18,76],
  [18,84],
  [19,36],
  [19,46],
  [19,68],
  [20,32],
  [20,48],
  [20,52],
  [20,56],
  [20,64],
  [22,66],
  [23,54],
  [24,70],
  [24,92],
  [24,93],
  [24,120],
  [25,85]
])
```

We can extract the two columns using slicing:

```
speed = data[:, 0]
stopping_distance = data[:, 1]
```

Let's have a look at the data:



It seems like a linear model can do a decent job of predicting the stopping distance. Simple Linear Regression finds a straight line that predicts the variable of interest based on a single predictor/feature.

Time to build the model using the Keras API:

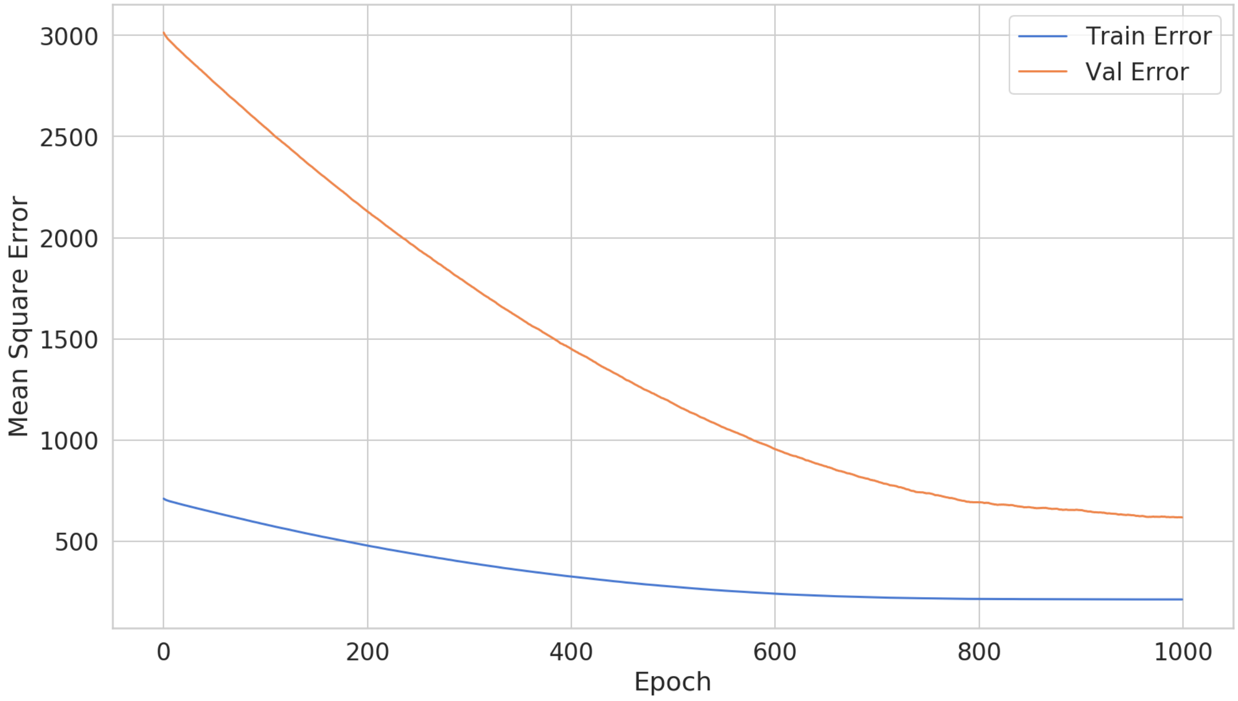
```
lin_reg = keras.Sequential([
  layers.Dense(1, activation='linear', input_shape=[1]),
])

optimizer = tf.keras.optimizers.RMSprop(0.001)

lin_reg.compile(
  loss='mse',
  optimizer=optimizer,
  metrics=['mse']
)
```

We're using the Sequential API with a single layer - 1 parameter with linear activation. We'll try to minimize the Mean squared error[8] during training.

And for the training itself:

```
history = lin_reg.fit(
  x=speed,
  y=stopping_distance,
  shuffle=True,
  epochs=1000,
  validation_split=0.2,
  verbose=0
)
```

We're breaking any ordering issues by shuffling the data and reserving 20% for validation. Let's have a look at the training process:



The model is steadily improving during training. That's a good sign. What can we do with a more complex model?

Simple Neural Network Model

Keras (and TensorFlow) was designed as a tool to build Neural Networks. Turns out, Neural Networks are good when a linear model isn't enough. Let's create one:

[8] https://en.wikipedia.org/wiki/Mean_squared_error

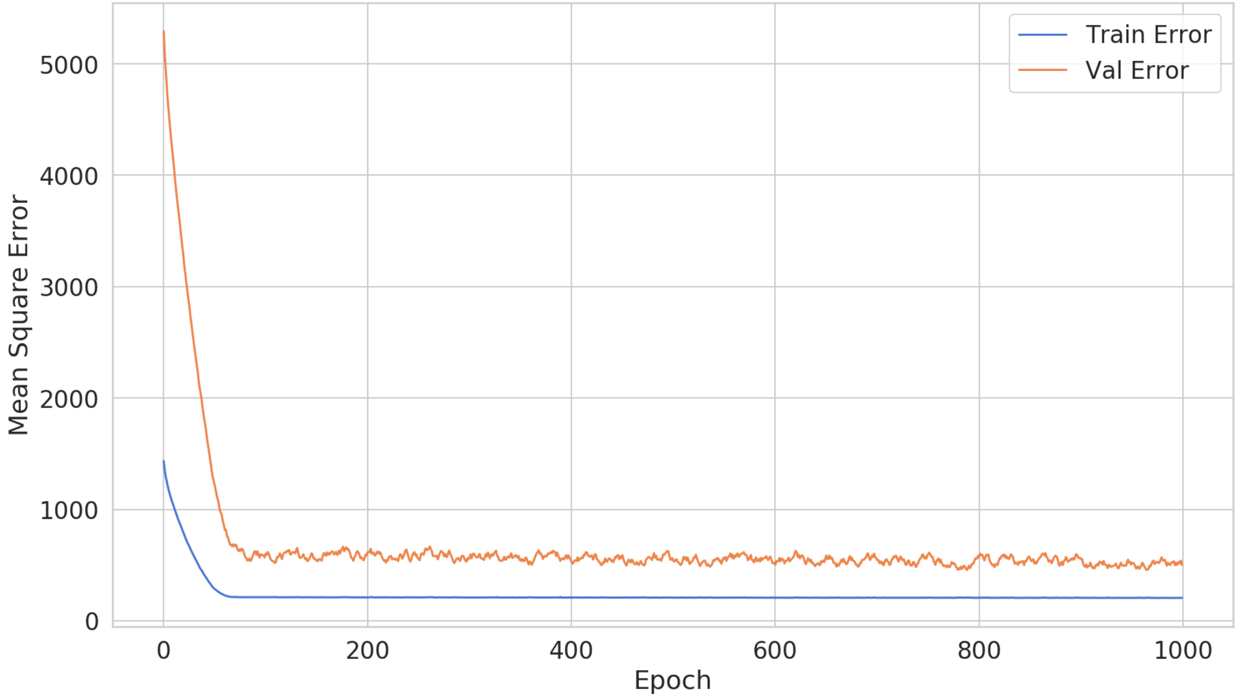
```python
def build_neural_net():
  net = keras.Sequential([
    layers.Dense(32, activation='relu', input_shape=[1]),
    layers.Dense(16, activation='relu'),
    layers.Dense(1),
  ])

  optimizer = tf.keras.optimizers.RMSprop(0.001)

  net.compile(loss='mse',
              optimizer=optimizer,
              metrics=['mse', 'accuracy'])

  return net
```

Things look similar, except for the fact that we stack multiple layers on top of each other. We're also using a different activation function - ReLU[9].

Training this model looks exactly the same:

```python
net = build_neural_net()

history = net.fit(
  x=speed,
  y=stopping_distance,
  shuffle=True,
  epochs=1000,
  validation_split=0.2,
  verbose=0
)
```

[9] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)



Seems like we ain't making much progress after epoch 200 or so. Can we not waste our time waiting for the whole training to complete?

Early Stopping

Sure, you can stop the training process manually at say epoch 200. But what if you train another model? What if you obtain more data?

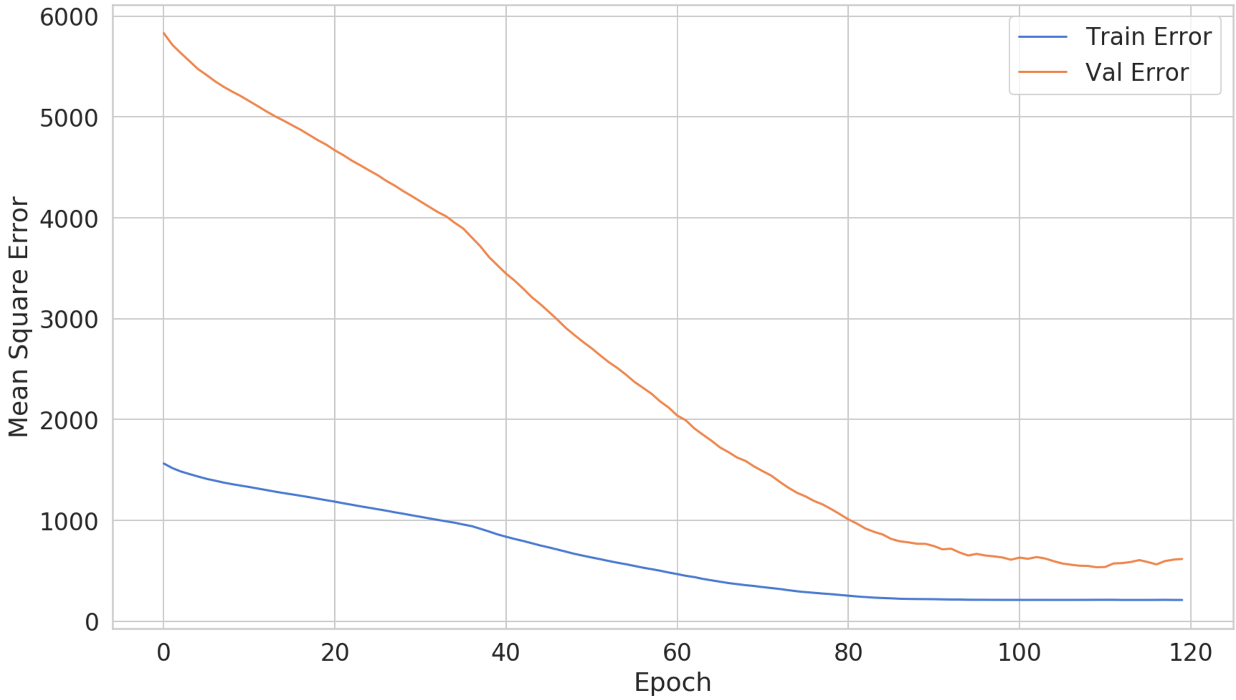
You can use the built-in callback EarlyStopping[10] to halt the training when some metric (e.g. the validation loss) stops improving. Let's see how we can use it:

```
early_stop = keras.callbacks.EarlyStopping(
  monitor='val_loss',
  patience=10
)
```

We want to monitor the validation loss. We'll observe for improvement for 10 epochs before stopping. Let's see how we can use it:

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

```
net = build_neural_net()

history = net.fit(
  x=speed,
  y=stopping_distance,
  shuffle=True,
  epochs=1000,
  validation_split=0.2,
  verbose=0,
  callbacks=[early_stop]
)
```



Effectively, we've cut down the number of training epochs to ∼120. Is this going to work every time that well? Not really. Using early stopping introduces yet another hyperparameter that you need to consider when training your model. Use it cautiously.

Now your model is ready for the real world. How can you store it for later use?

Save/Restore Model

You can save the complete model (including weights) like this:

```
net.save('simple_net.h5')
```

And load it like that:

```
simple_net = keras.models.load_model('simple_net.h5')
```

You can use this mechanism to deploy your model and use it in production (for example).

Conclusion

You did it! You now know (a tiny bit) TensorFlow 2! Let's recap what you've learned:

- How to install TensorFlow 2
- What is a Tensor
- Doing Tensor math
- Using probability distributions and sampling
- Build a Simple Linear Regression model
- Build a Simple Neural Network model
- Save/restore a model

Run the complete code in your browser[11]

Stay tuned for more :)

References

- TensorFlow 2.0 released[12]
- TensorFlow 2.0 on GitHub[13]
- Effective TensorFlow 2.0[14]

[11] https://colab.research.google.com/drive/1HkG7HYS1-IFAYbECZ0zleBWA3Xi4DKIm
[12] https://medium.com/tensorflow/tensorflow-2-0-is-now-available-57d706c2a9ab
[13] https://github.com/tensorflow/tensorflow/releases/tag/v2.0.0
[14] https://www.tensorflow.org/guide/effective_tf2

Build Your First Neural Network

> TL;DR Build and train your first Neural Network model using TensorFlow 2. Use the model to recognize clothing type from images.

Ok, I'll start with a secret—I am THE fashion wizard (as long as we're talking tracksuits). Fortunately, there are ways to get help, even for someone like me!

Can you imagine a really helpful browser extension for "fashion accessibility"? Something that tells you what the type of clothing you're looking at.

After all, I really need something like this. I found out nothing like this exists, without even searching for it. Let's make a Neural Network that predicts clothing type from an image!

Here's what we are going to do:

1. Install TensorFlow 2
2. Take a look at some fashion data
3. Transform the data, so it is useful for us
4. Create your first Neural Network in TensorFlow 2
5. Predict what type of clothing is showing on images your Neural Network haven't seen

Setup

With TensorFlow 2 just around the corner (not sure how far along that corner is thought) making your first Neural Network has never been easier (as far as TensorFlow goes).

But what is TensorFlow[15]? Machine Learning platform (really Google?) created and open sourced by Google. Note that TensorFlow is not a special purpose library for creating Neural Networks, although it is primarily used for that purpose.

So, what TensorFlow 2 has in store for us?

> TensorFlow 2.0 focuses on simplicity and ease of use, with updates like eager execution, intuitive higher-level APIs, and flexible model building on any platform

Alright, let's check those claims and install TensorFlow 2 from your terminal:

[15] https://www.tensorflow.org/overview

```
pip install tensorflow-gpu==2.0.0-alpha0
```

Fashion data

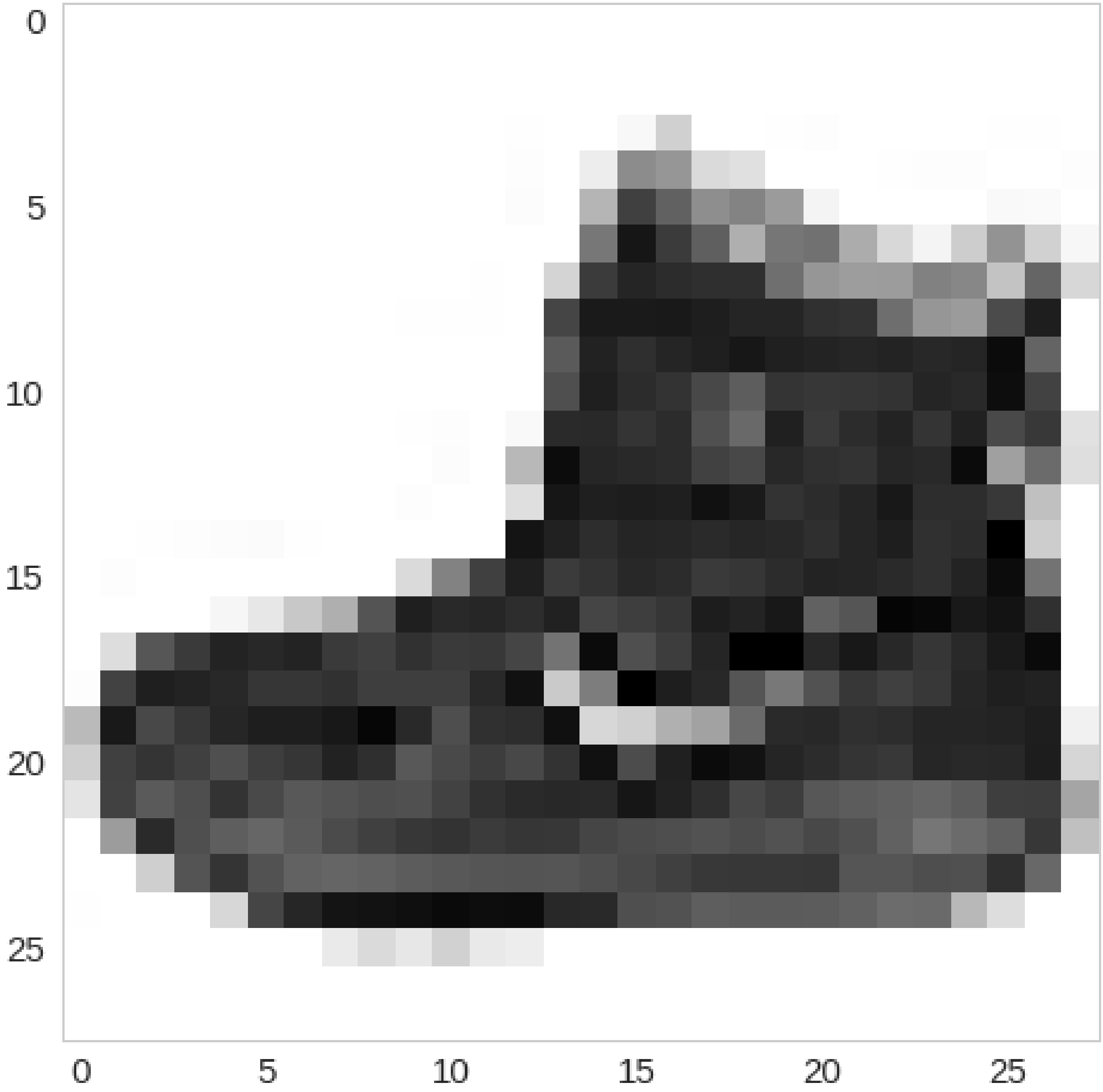
Your Neural Network needs something to learn from. In Machine Learning that something is called datasets. The dataset for today is called Fashion MNIST[16].

> `Fashion-MNIST` is a dataset of Zalando's article images[17] — consisting of a training set of *60,000* examples and a test set of *10,000* examples. Each example is a *28x28* grayscale image, associated with a label from *10* classes.

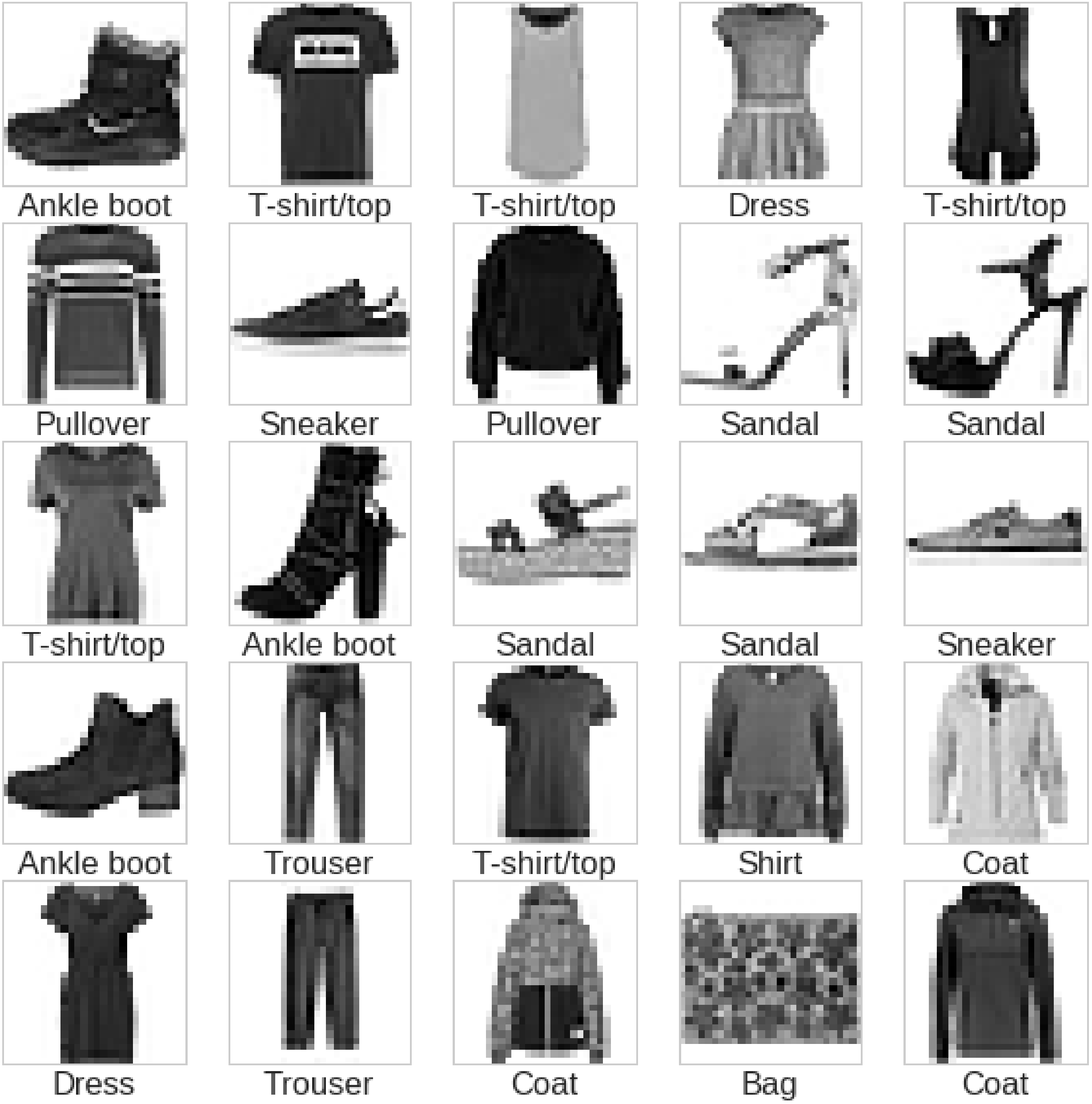
In other words, we have *70,000* images of *28* pixels width and *28* pixels height in greyscale. Each image is showing one of *10* possible clothing types. Here is one:

[16] https://github.com/zalandoresearch/fashion-mnist

[17] https://jobs.zalando.com/en/



Here are some images from the dataset along with the clothing they are showing:



Here are all different types of clothing:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Now that we got familiar with the data we have let's make it usable for our Neural Network.

Data Preprocessing

Let's start with loading our data into memory:

```
import tensorflow as tf
from tensorflow import keras

(x_train, y_train), (x_val, y_val) = keras.datasets.fashion_mnist.load_data()
```

Fortunately, TensorFlow has the dataset built-in, so we can easily obtain it.

Loading it gives us 4 things:

`x_train` — image (pixel) data for *60,000* clothes. Used for *training* our model.

`y_train` — classes (clothing type) for the clothing above. Used for *training* our model.

`x_val` — image (pixel) data for *10,000* clothes. Used for *testing/validating* our model.

`y_val` — classes (clothing type) for the clothing above. Used for *testing/validating* our model.

Now, your Neural Network can't really see images as you do. But it can understand numbers. Each data point of each image in our dataset is pixel data—a number between 0 and 255. We would like that data to be transformed (Why? While the truth is more nuanced, one can say it helps with training a better model) in the range 0–1. How can we do it?

We will use the Dataset[18] from TensorFlow to prepare our data:

[18] https://www.tensorflow.org/api_docs/python/tf/data/Dataset

```
def preprocess(x, y):
  x = tf.cast(x, tf.float32) / 255.0
  y = tf.cast(y, tf.int64)

  return x, y

def create_dataset(xs, ys, n_classes=10):
  ys = tf.one_hot(ys, depth=n_classes)
  return tf.data.Dataset.from_tensor_slices((xs, ys)) \
    .map(preprocess) \
    .shuffle(len(ys)) \
    .batch(128)
```

Let's unpack what is happening here. What does tf.one_hot do? Let's say you have the following vector:

`[1, 2, 3, 1]`

Here is the one-hot encoded version of it:

```
[
    [1, 0, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 0]
]
```

It puts 1 at the index position of the number and 0 everywhere else.

We create *Dataset* from the data using from_tensor_slices[19] and divide each pixel of the images by 255 to scale it in the 0−1 range.

Then we use shuffle[20] and batch[21] to convert the data into chunks.

Why shuffle the data, though? We don't want our model to make predictions based on the order of the training data, so we just shuffle it.

I am truly sorry for this bad joke[22]

Create your first Neural Network

You're doing great! It is time for the fun part, use the data to create your first Neural Network.

[19] https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices
[20] https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle
[21] https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch
[22] https://www.youtube.com/watch?v=KQ6zr6kCPj8

```
train_dataset = create_dataset(x_train, y_train)
val_dataset = create_dataset(x_val, y_val)
```

Build your Neural Network using Keras layers

They say TensorFlow 2 has an easy High-level API, let's take it for a spin:

```
model = keras.Sequential([
    keras.layers.Reshape(
      target_shape=(28 * 28,), input_shape=(28, 28)
    ),
    keras.layers.Dense(
      units=256, activation='relu'
    ),
    keras.layers.Dense(
      units=192, activation='relu'
    ),
    keras.layers.Dense(
      units=128, activation='relu'
    ),
    keras.layers.Dense(
      units=10, activation='softmax'
    )
])
```

Turns out the High-level API is the old Keras[23] API which is great.

Most Neural Networks are built by "stacking" layers. Think pancakes or lasagna. Your first Neural Network is really simple. It has 5 layers.

The first (Reshape[24]) layer is called an input layer and takes care of converting the input data for the layers below. Our images are 28*28=784 pixels. We're just converting the 2D 28x28 array to a 1D 784 array.

All other layers are Dense[25] (interconnected). You might notice the parameter **units**, it sets the number of neurons for each layer. The **activation** parameter specifies a function that decides whether "the opinion" of a particular neuron, in the layer, should be taken into account and to what degree. There are a lot of activation functions one can use.

The last (output) layer is a special one. It has 10 neurons because we have 10 different types of clothing in our data. You get the predictions of the model from this layer.

[23] https://keras.io/
[24] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Reshape
[25] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense

Train your model

Right now your Neural Network is plain dumb. It is like a shell without a soul (good that you get that). Let's train it using our data:

```
model.compile(
  optimizer='adam',
  loss=tf.losses.CategoricalCrossentropy(from_logits=True),
  metrics=['accuracy']
)

history = model.fit(
    train_dataset.repeat(),
    epochs=10,
    steps_per_epoch=500,
    validation_data=val_dataset.repeat(),
    validation_steps=2
)
```

Training a Neural Network consists of deciding on objective measurement of accuracy and an algorithm that knows how to improve on that.

TensorFlow allows us to specify the optimizer algorithm we're going to use — Adam[26] and the measurement (loss function) — CategoricalCrossentropy[27] (we're choosing/classifying 10 different types of clothing). We're measuring the accuracy of the model during the training, too!

The actual training takes place when the fit method is called. We give our training and validation data to it and specify how many epochs we're training for. During one training epoch, all data is shown to the model.

Here is a sample result of our training:

```
Epoch 1/10 500/500 [==============================] - 9s 18ms/step - loss: 1.7340 - \
accuracy: 0.7303 - val_loss: 1.6871 - val_accuracy: 0.7812
Epoch 2/10 500/500 [==============================] - 6s 12ms/step - loss: 1.6806 - \
accuracy: 0.7807 - val_loss: 1.6795 - val_accuracy: 0.7812
...
```

I got ~82% accuracy on the validation set after 10 epochs. Lets profit from our model!

Making predictions

Now that your Neural Network "learned" something lets try it out:

[26] https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer

[27] https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy

```
predictions = model.predict(val_dataset)
```

Here is a sample prediction:

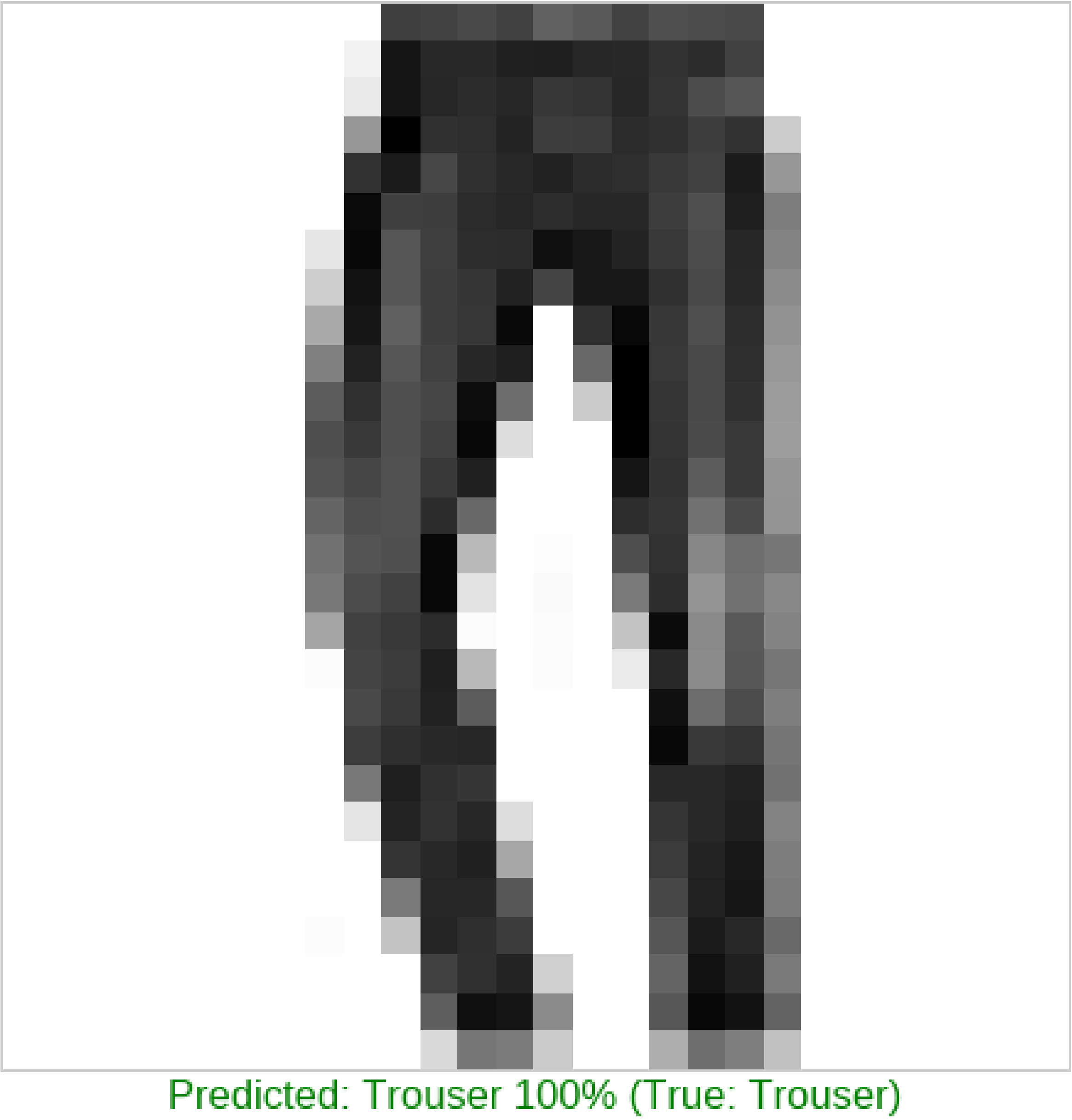
```
array([
 1.8154810e-07,
 1.0657334e-09,
 9.9998713e-01,
 1.1928002e-05,
 2.9766360e-08,
 4.0670972e-08,
 2.5100772e-07,
 4.5147233e-11,
 2.9812568e-07,
 3.5224868e-11
], dtype=float32)
```

Recall that we have 10 different clothing types. Our model outputs a probability distribution about how likely each clothing type is shown on an image. To make a decision, we can get the one with the highest probability:

```
np.argmax(predictions[0])
```

2

Here is one correct and one wrong prediction from our model:


Predicted: Trouser 100% (True: Trouser)

Predicted: Trouser 100% (True: Ankle boot)

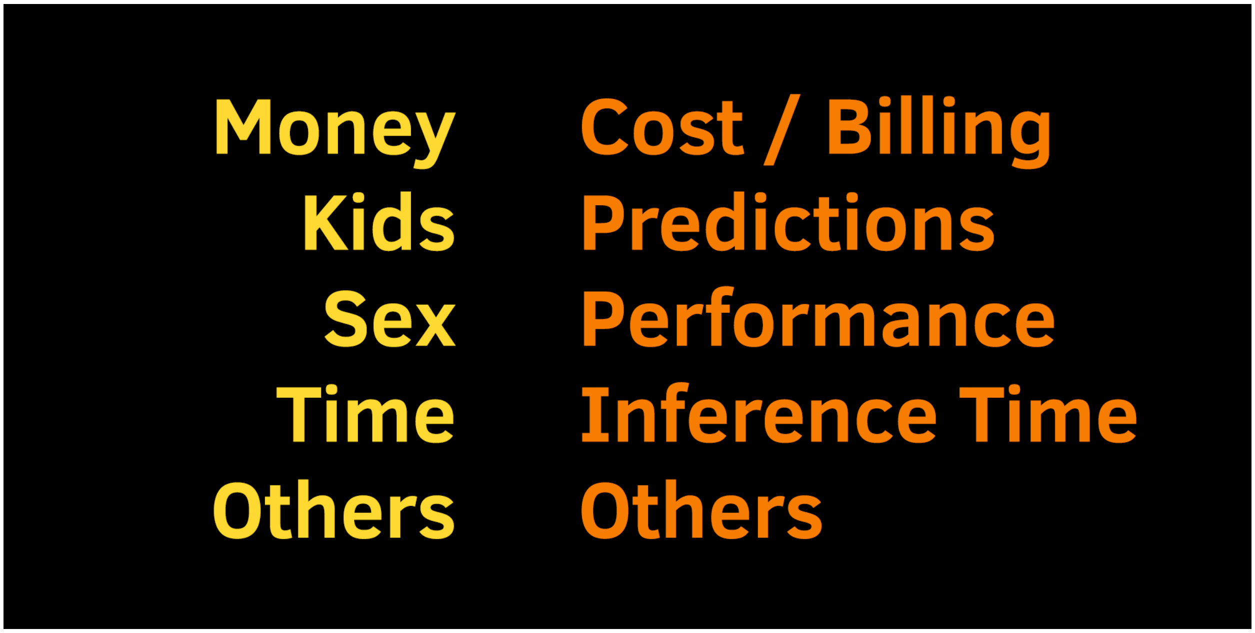
Conclusion

Alright, you got your first Neural Network running and made some predictions! You can take a look at the Google Colaboratory Notebook (including more charts) here:

Google Colaboratory Notebook[28]

[28] https://colab.research.google.com/drive/1ctyhVlD9Y85KTBma1X9Zf35Q0ha9PCaP

One day you might realize that your relationship with Machine Learning is similar to marriage. The problems you might encounter are similar, too! What Makes Marriages Work by John Gottman, Nan Silver[29] lists 5 problems marriages have: "Money, Kids, Sex, Time, Others". Here are the Machine Learning counterparts:



Money	Cost / Billing
Kids	Predictions
Sex	Performance
Time	Inference Time
Others	Others

Shall we tackle them together?

[29] https://www.psychologytoday.com/intl/articles/199403/what-makes-marriage-work

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[30]

The complete project on GitHub[31]

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[32] 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.

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

Load data

The data comes from Airbnb Open Data and it is hosted on Kaggle[33]

> 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:

[33] 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:

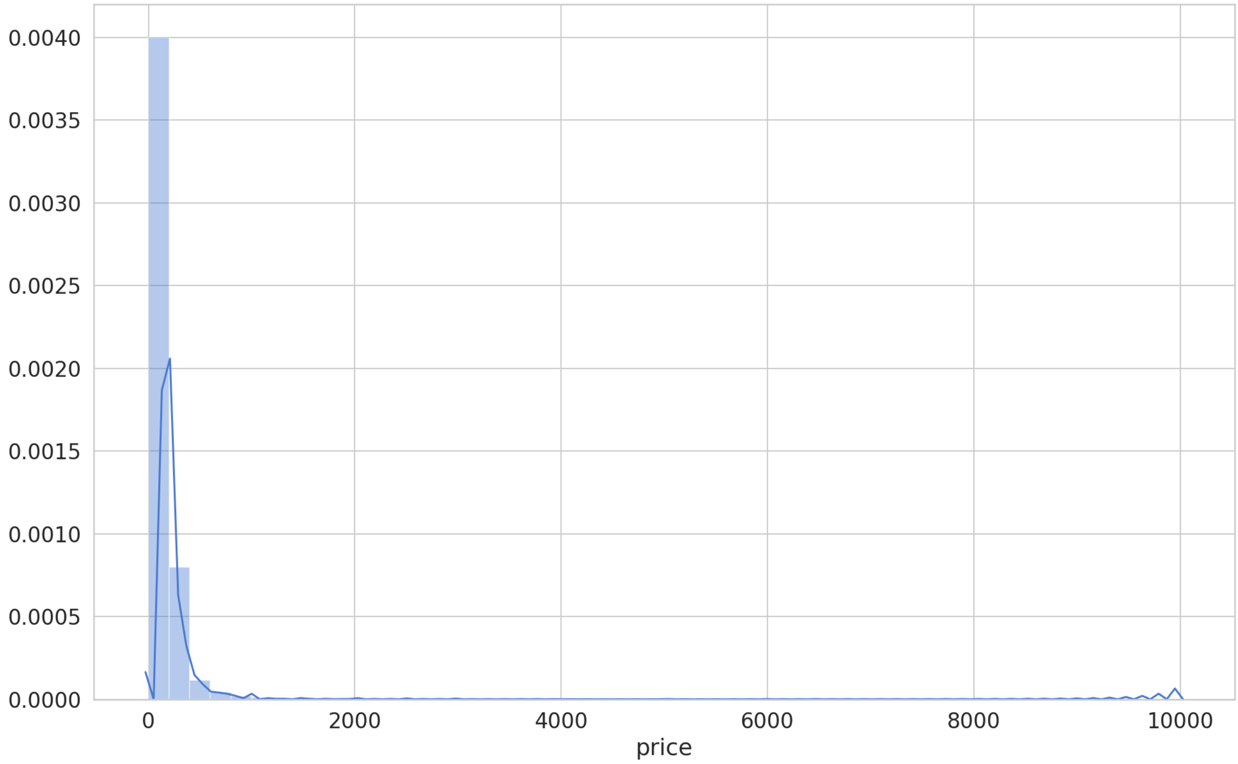
```
df.shape
```

```
(48895, 16)
```

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

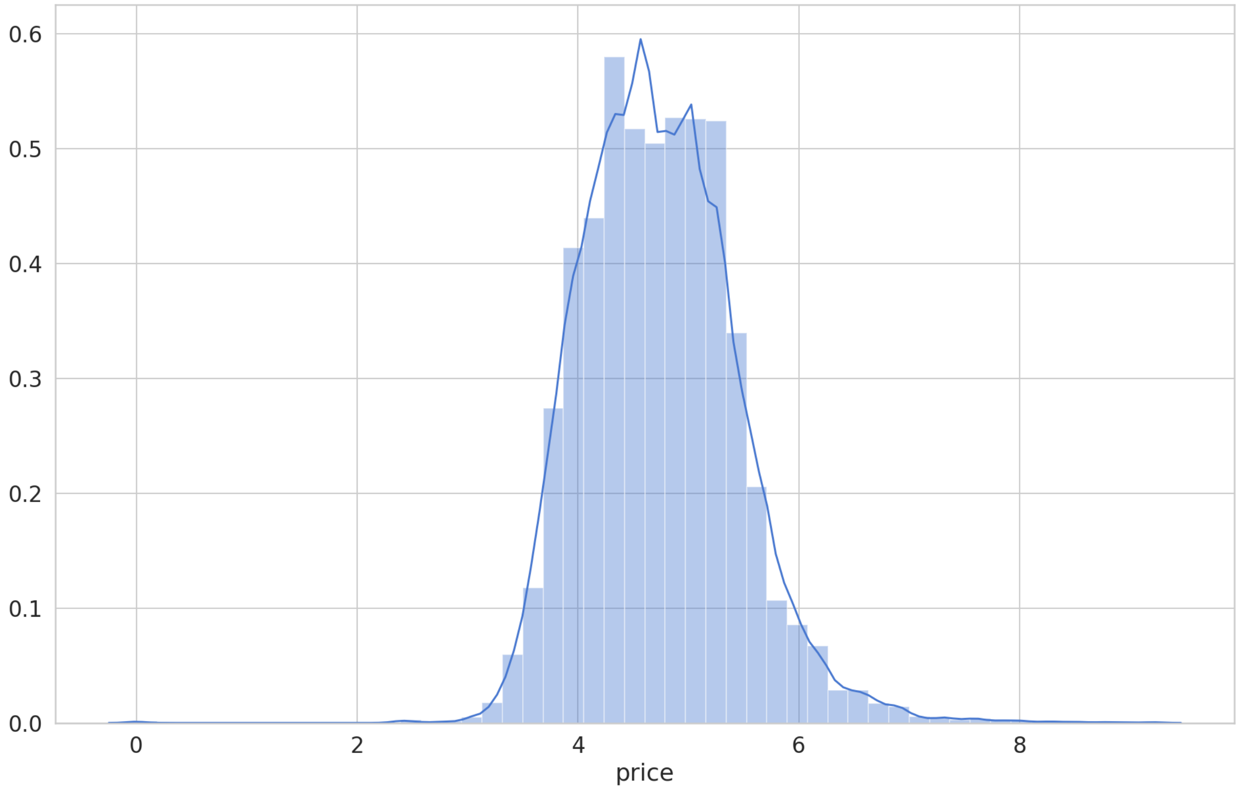
Let's start with the variable we're trying to predict `price`. To plot the distribution, we'll use `distplot()`:

```
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))
```

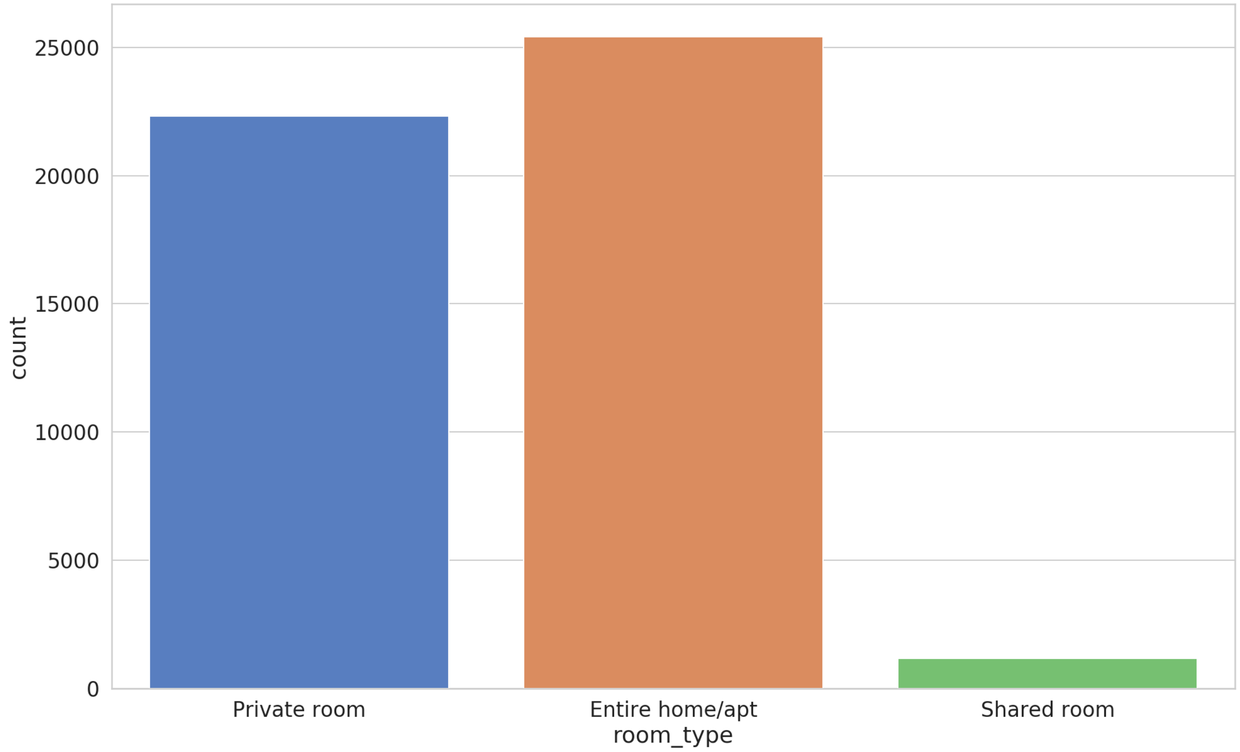


This looks more like a normal distribution. Turns out this might help your model better learn the data[34]. You'll have to remember to preprocess the data before training and predicting.

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

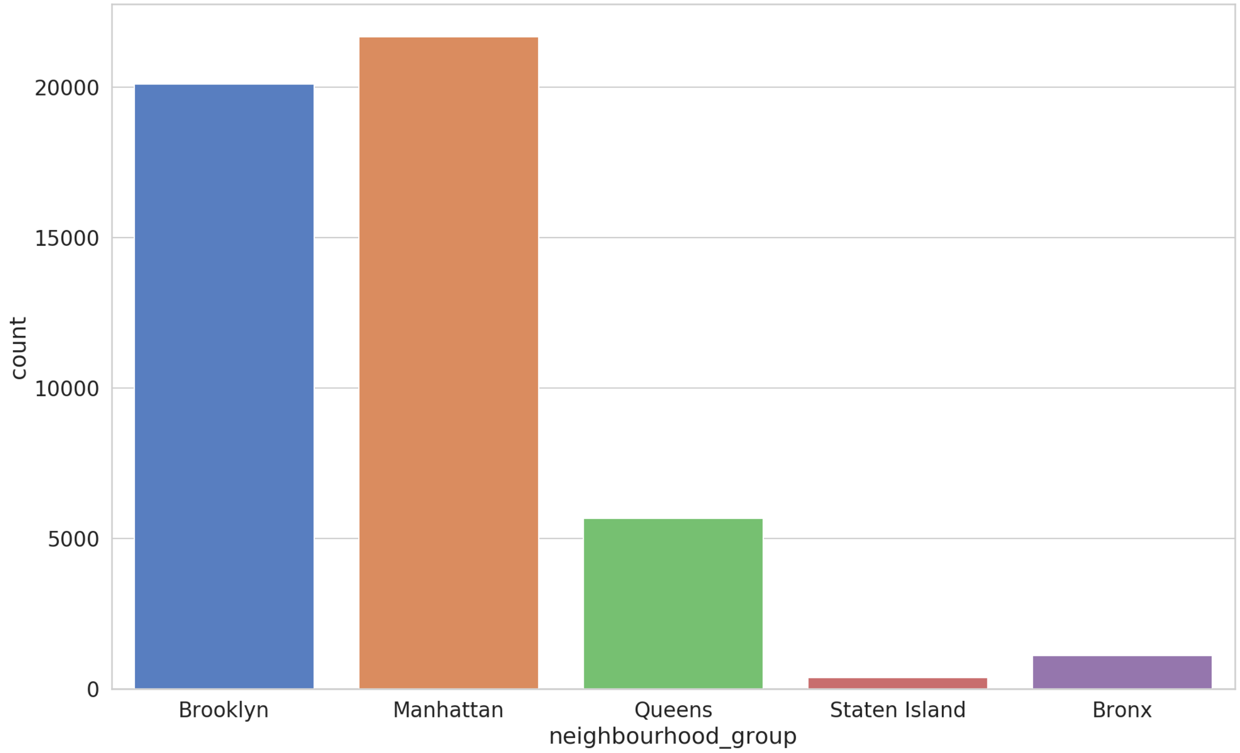
```
sns.countplot(x='room_type', data=df)
```

[34] 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?

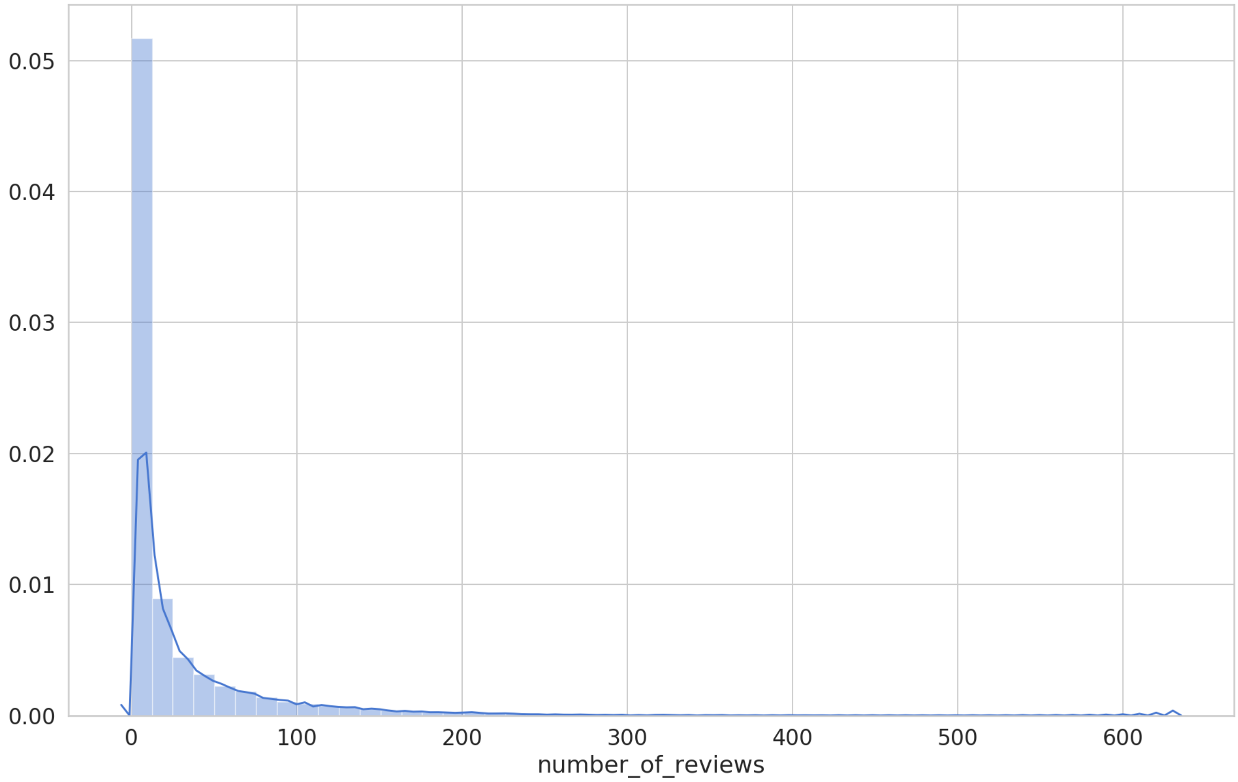
```
sns.countplot(x='neighbourhood_group', data=df)
```



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[35] (it has a fat tail). This one seems to follow a Power law[36] (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[37]

Computing Pearson correlation coefficient[38] between a pair of features is easy:

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

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

[35] https://en.wikipedia.org/wiki/Power_law
[36] https://en.wikipedia.org/wiki/Power_law
[37] https://en.wikipedia.org/wiki/Correlation_does_not_imply_causation
[38] 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[39]. Specifically, we'll do min-max normalization and scale the features in the 0-1 range. Luckily, the MinMaxScaler[40] 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[41] 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[42]. To get a feel of what it does, we'll use OneHotEncoder[43] from scikit-learn:

[39] https://en.wikipedia.org/wiki/Feature_scaling
[40] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
[41] https://arxiv.org/abs/1502.03167
[42] https://en.wikipedia.org/wiki/One-hot
[43] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

```python
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()`[44]:

```python
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:

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

Finally, we'll transform our data:

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

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

[44] 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[45] 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[46] framework.

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

Error measurement

We'll use Mean Squared Error[47] which measures the difference between average squared predicted and true values:

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

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[48] is the official high-level API for TensorFlow[49]. In short, it allows you to build complex models using a sweet interface. Let's build a model with it:

[45] https://www.airbnb.com/
[46] https://en.wikipedia.org/wiki/Regression_analysis
[47] https://en.wikipedia.org/wiki/Mean_squared_error
[48] https://keras.io/
[49] 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[50].

How can we specify the error metric?

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

The `compile()`[51] 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()`[52]:

```
BATCH_SIZE = 32

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

history = model.fit(
```

[50] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout
[51] https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile
[52] 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:

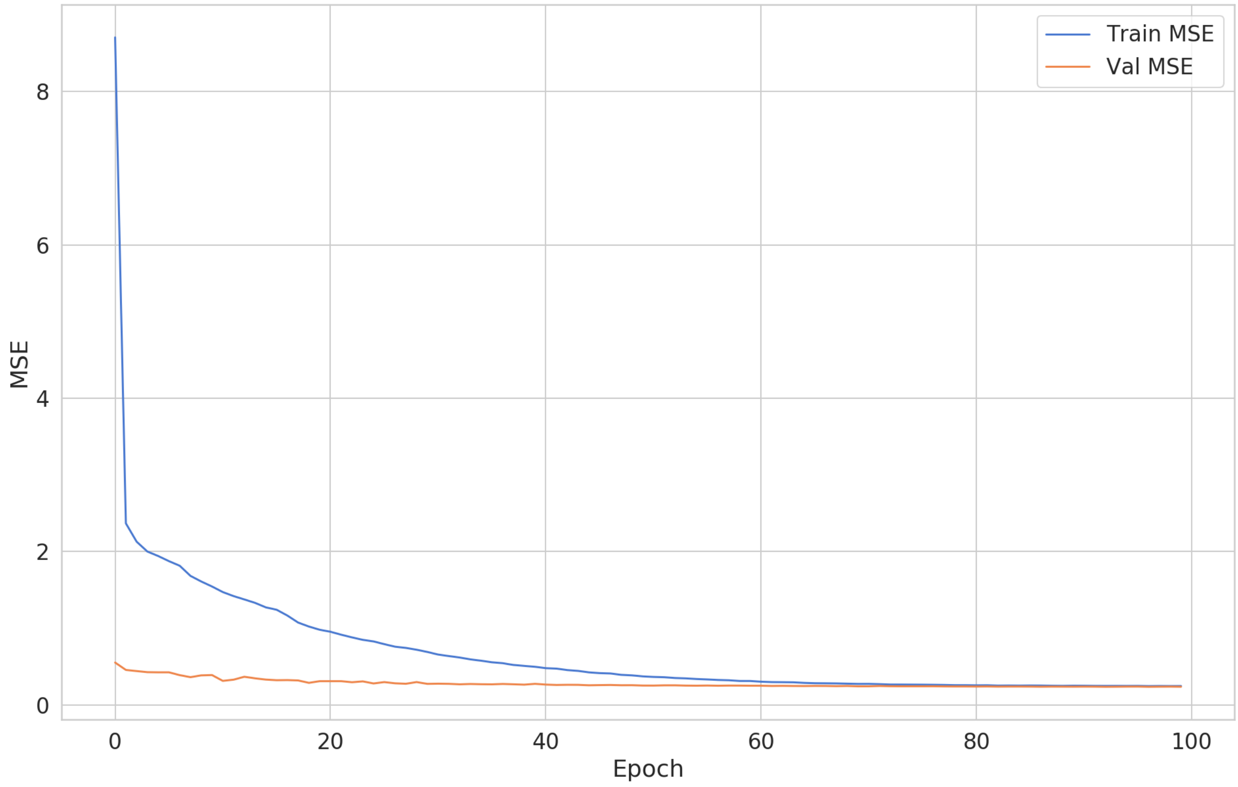
- shuffle - random sort the data
- epochs - number of training cycles
- validation_split - use some percent of the data for measuring the error and not during training
- batch_size - the number of training examples that are fed at a time to our model
- callbacks - we use EarlyStopping[53] to prevent 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:

[53] 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:

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

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

```
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)[54] 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[55]. 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[56] is to use joblib[57]. Saving the model architecture and weights of a Keras model is done with the `save()`[58] method.

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

[54] https://en.wikipedia.org/wiki/Root-mean-square_deviation
[55] https://en.wikipedia.org/wiki/Coefficient_of_determination
[56] https://scikit-learn.org/stable/modules/model_persistence.html#persistence-example
[57] https://joblib.readthedocs.io/en/latest/
[58] https://www.tensorflow.org/api_docs/python/tf/keras/Sequential#save

```
from google.colab import files

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

Build REST API

Building a REST API[59] 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[60] 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:

```
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[61]

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

[59] https://en.wikipedia.org/wiki/Representational_state_transfer
[60] https://www.fullstackpython.com/flask.html
[61] 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[62]:

> 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[63]
- Google Cloud SDK installed[64]

Here is the complete `app.yaml` config:

[62] https://cloud.google.com/appengine/
[63] https://cloud.google.com/compute/
[64] 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[65]

The complete project on GitHub[66]

[65] https://colab.research.google.com/drive/1YxCmQb2YKh7VuQ_XgPXhEeIM3LpjV-mS

[66] https://github.com/curiousily/Deploy-Keras-Deep-Learning-Model-with-Flask

References

- Joblib - running Python functions as pipeline jobs[67]
- Flask - lightweight web application framework[68]
- Building a simple Keras + deep learning REST API[69]

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

Fundamental Machine Learning Algorithms

> TL;DR Overview of fundamental classification and regression learning algorithms. Learn when should you use each and what data preprocessing is required. Each algorithm is presented along with an example done in scikit-learn.

This guide explores different supervised learning algorithms[70], sorted by increasing complexity (measured by the number of model parameters and hyperparameters). I would strongly suggest you start with the simpler ones when working on a new project/problem.

But why not just use Deep Neural Networks for everything? You can, and maybe you should. But simplicity can go a long way before the need for ramping up the complexity in your project. It is also entirely possible to not be able to tune your Neural Net to beat some of the algorithms described here.

You're going to learn about:

- Linear Regression
- Logistic Regression
- k-Nearest Neighbors
- Naive Bayes
- Decision Trees
- Support Vector Machines

Run the complete notebook in your browser[71]

The complete project on GitHub[72]

What Makes a Learning Algorithm?

In their essence, supervised Machine Learning algorithms learn a mapping function f that maps the data features X to labels y. The goal is to make the mapping as accurate as possible. We can define the problem as:

$$y = f(X) + e$$

[70] https://en.wikipedia.org/wiki/Supervised_learning
[71] https://colab.research.google.com/drive/1-_wQbYW-KqDNMkT9iZ-d2KWVHyR6d0nn
[72] https://github.com/curiousily/Deep-Learning-For-Hackers

where e is an irreducible error. That error is independent of the data and can't be lowered using the data (hence the name).

The problem of finding the function f is notoriously difficult. In practice, you'll be content with a good approximation. There are many ways to get the job done. What do they have in common?

Components of a Learning Algorithm

- Loss functions
- Optimizer that tries to minimize the loss function

The *Loss Function*[73] outputs a numerical value that shows how "bad" your model predictions are. The closer the value is to 0, the better the predictions.

The optimizer's job is to find the best possible values for the model parameters that minimize the loss function. This is done with the help of the training data and an algorithm that searches for the parameter values.

Gradient Descent[74] is the most commonly used algorithm for optimization. It finds a local minimum of a function by starting at a random point and takes steps in a direction and size given by the gradient.

Our Data

We'll use the Auto Data Set[75] to create examples for various classification and regression algorithms.

> Gas mileage, horsepower, and other information for 392 vehicles. This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University. The dataset was used in the 1983 American Statistical Association Exposition.

Let's download the data and load it into a Pandas data frame:

```
!gdown --id 16VDAc-x1fGa2lpsI8xtLHK6z_3m36JBx --output auto.csv
```

```
auto_df = pd.read_csv("auto.csv", index_col=0)
auto_df.shape
```

[73] https://en.wikipedia.org/wiki/Loss_function
[74] https://en.wikipedia.org/wiki/Gradient_descent
[75] https://vincentarelbundock.github.io/Rdatasets/doc/ISLR/Auto.html

```
(392, 9)
```

We have 392 vehicles and we'll use this subset of the features:

- *mpg* - miles per gallon
- *horsepower* - Engine horsepower
- *weight* - Vehicle weight (lbs.)
- *acceleration* - Time to accelerate from 0 to 60 mph (sec.)
- *origin* - Origin of car (1. American, 2. European, 3. Japanese)

We have no missing data.

Data Preprocessing

We're going to define two helper functions that prepare a classification and a regression dataset based on our data. But first, we're going to add a new feature that specifies whether a car is American made or not:

```
auto_df['is_american'] = (auto_df.origin == 1).astype(int)
```

We're going to use the StandarScaler[76] to scale our datasets:

```
from sklearn.preprocessing import StandardScaler

def create_regression_dataset(
    df,
    columns=['mpg', 'weight', 'horsepower']
):

  all_columns = columns.copy()
  all_columns.append('acceleration')

  reg_df = df[all_columns]
  reg_df = StandardScaler().fit_transform(reg_df[all_columns])
  reg_df = pd.DataFrame(reg_df, columns=all_columns)

  return reg_df[columns], reg_df.acceleration

def create_classification_dataset(df):
  columns = ['mpg', 'weight', 'horsepower']
```

[76] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

```

  X = df[columns]
  X = StandardScaler().fit_transform(X)
  X = pd.DataFrame(X, columns=columns)

  return X, df.is_american
```

Evaluation

We’re going to use k-fold cross validation[77] to evaluate the performance of our models. *Note that this guide is NOT benchmarking model performance.* Here are the definitions of our evaluation functions:

```
from sklearn.model_selection import KFold, cross_val_score

def eval_model(model, X, y, score):
  cv = KFold(n_splits=10, random_state=RANDOM_SEED)
  results = cross_val_score(model, X, y, cv=cv, scoring=score)
  return np.abs(results.mean())

def eval_classifier(model, X, y):
  return eval_model(model, X, y, score="accuracy")

def eval_regressor(model, X, y):
  return eval_model(model, X, y, score="neg_mean_squared_error")
```

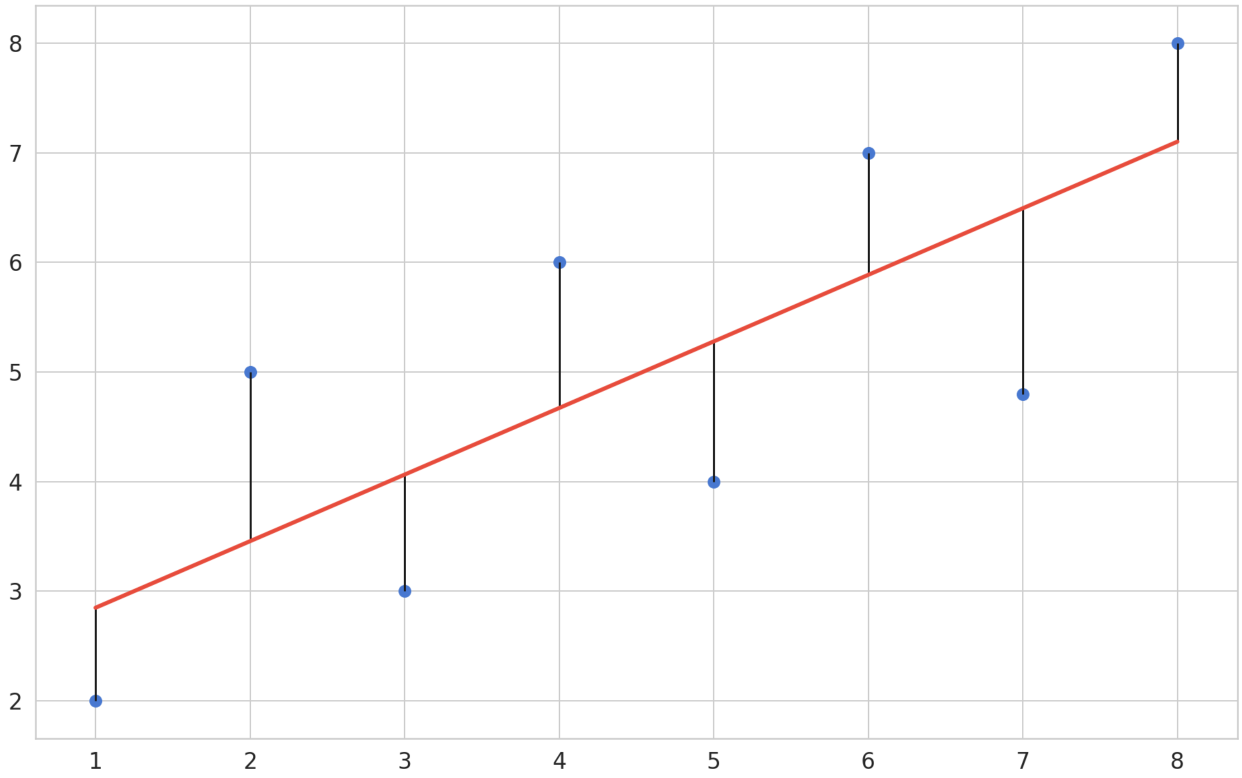
We are using accuracy (percent of correctly predicted examples) as a metric for our classification examples and mean squared error (explained below) for the regression examples.

Linear Regression

Linear Regression[78] tries to build a line that can best describe the relationship between two variables X and Y. That line is called “best-fit” and is closest to the points (x_i, y_i).

[77] https://en.wikipedia.org/wiki/Cross-validation_(statistics)#k-fold_cross-validation

[78] https://en.wikipedia.org/wiki/Linear_regression



Y is known as the *dependent variable* and it is *continious* - e.g. number of sales, price, weight. This is the variable which values we're trying to predict. X is known as explanatory (or independent) variable. We use this variable to predict the value of Y. *Note that we're assuming a linear relationship between the variables.*

Definition

Our dataset consists of m labeled examples (x_i, y_i), where x_i is $\mathbb{D}$-dimensional feature vector, $y_i \in \mathbb{R}$ and every feature $x_i^j \in \mathbb{R}, j = 1, \ldots, \mathbb{D}$. We want to build a model that predicts unknown y for a given x. Our model is defined as:

$$f_{w,b}(x) = wx + b$$

where w and b are parameters of our model that we'll learn from the data. w defines the *slope* of the model, while b defines the *intercept* point with the vertical axis.

Making Predictions

Linear regression that makes the most accurate prediction has optimal values for the parameters w and b. Let's denote those as w^* and b^*. How can we find those values?

We'll use an objective metric that tells us how good the current values are. *Optimal parameter values will minimize that metric.*

The most used metric in such cases is Mean Squared Error(MSE)[79]. It is defined as:

$$MSE = L(x) = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - f_{w,b}(x^{(i)}))^2$$

The MSE measures how much the average model predictions vary from the correct values. The number is higher when the model is making "bad" predictions. *Model that makes perfect predictions has a MSE of 0.*

We've transformed the problem of finding optimal values for our parameters to minimizing MSE. We can do that using an optimization algorithm known as Stochastic Gradient Descent[80].

Simple Linear Regression

This type of Linear Regression uses a single feature to predict the target variable. Let's use the `horsepower` to predict car `acceleration`:

```python
from sklearn.linear_model import LinearRegression

X, y = create_regression_dataset(auto_df, columns=['horsepower'])

reg = LinearRegression()
eval_regressor(reg, X, y)
```

```
0.5283214994429212
```

Multiple Linear Regression

Of course, we can use more features to predict the `acceleration`. This is called *Multiple Linear Regression*. The training process looks identical (thanks to the nice interface that scikit-learn[81] provides):

[79] https://en.wikipedia.org/wiki/Mean_squared_error
[80] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[81] https://scikit-learn.org/stable/

```
X, y = create_regression_dataset(auto_df)

reg = LinearRegression()
eval_regressor(reg, X, y)
```

```
0.4351523357394419
```

The R^2 score has increased. Can we do better? How?

Ridge Regression

```
from sklearn.linear_model import Ridge

X, y = create_regression_dataset(auto_df)

reg = Ridge(alpha=0.0005, random_state=RANDOM_SEED)

eval_regressor(reg, X, y)
```

```
0.4351510356810997
```

When To Use Linear Regression?

Start with this algorithm when starting a regression problem. Useful when your features can be separated by a straight line.

Pros:

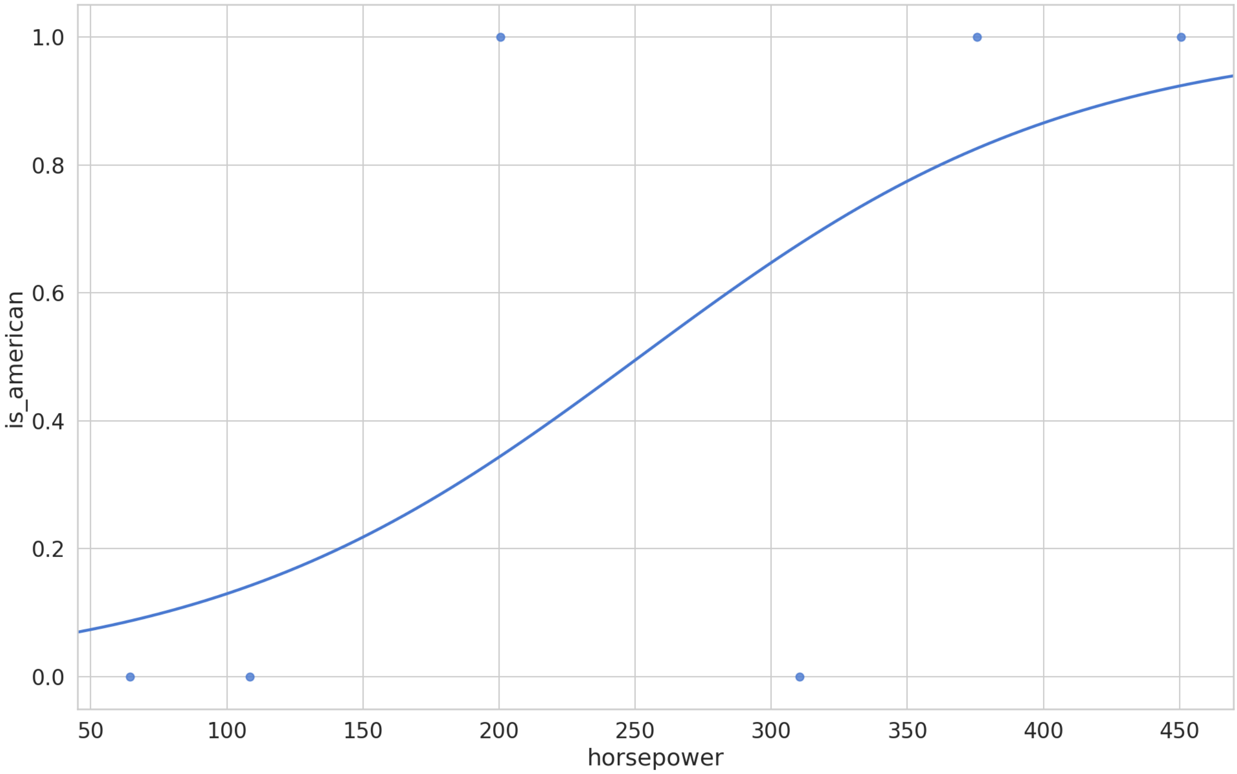
- Fast to train on large datasets
- Fast inference time
- Easy to understand/interpret the results

Cons:

- Need to scale numerical features
- Preprocess categorical data
- Can predict only linear relationships

Logistic Regression

Logistic Regression[82] has a similar formulation to Linear Regression (hence the name) but allows you to solve classification problems. The most common problem solved in practice is binary classification, so we'll discuss this application in particular.



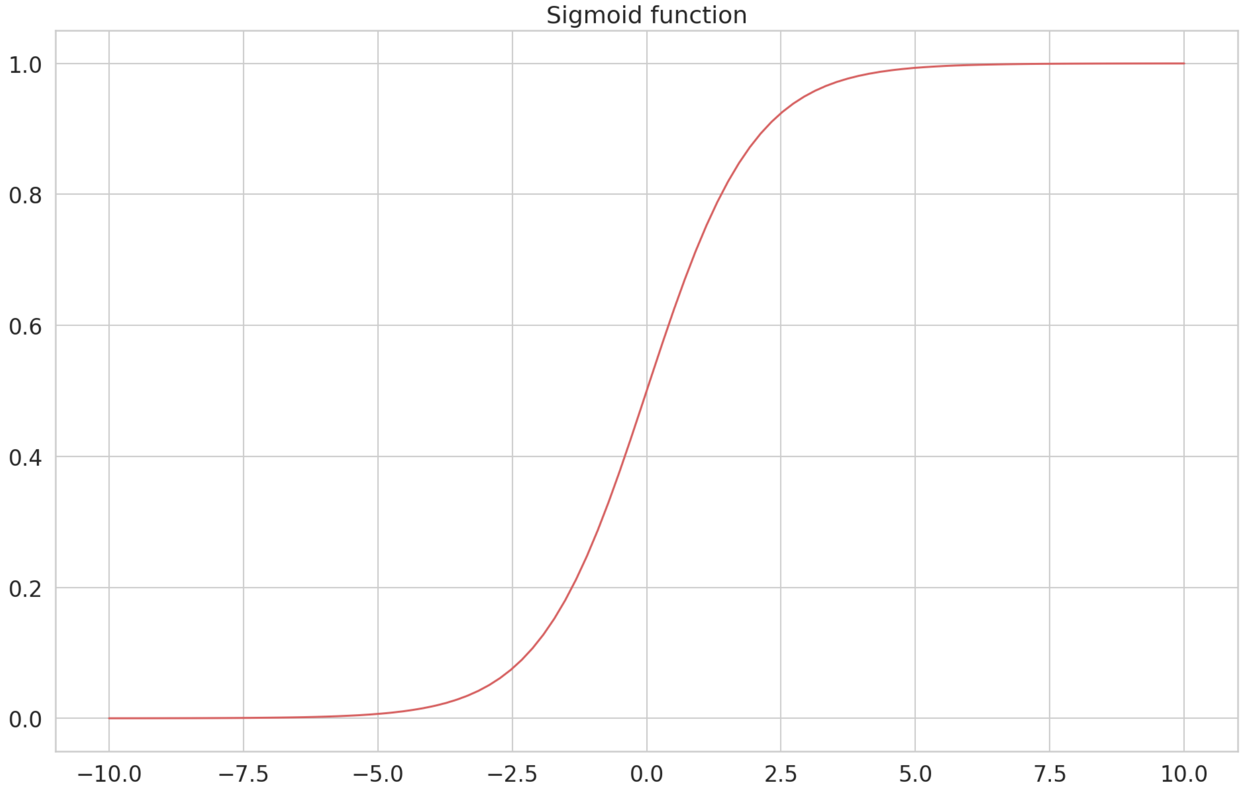
Making Predictions

We already have a way to make predictions with Linear Regression. The problem is that they are in $(-\infty, +\infty)$ interval. How can you use that to make true/false predictions?

If we map false to 0 and true to 1, we can use the Sigmoid function[83] to bound the domain to $(0, 1)$. It is defined by:

$$f(x) = \frac{1}{1 + e^{-x}}$$

[82] https://en.wikipedia.org/wiki/Logistic_regression
[83] https://en.wikipedia.org/wiki/Sigmoid_function



We can use the Sigmoid function and a predefined threshold (commonly set to 0.5) to map values larger than the threshold to a positive label; otherwise, it's negative.

Combining the Linear Regression equation with the Sigmoid function gives us:

$$f_{w,b}(x) = \frac{1}{1 + e^{-(wx+b)}}$$

Your next task is to find optimal parameter values for w^* and b^*. We can use the Log Loss[84] to measure how good our classifications are:

$$\text{Log Loss} = L(x) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log\, f_{w,b}(x) + (1 - y_i) \log{(1 - f_{w,b}(x))}]$$

Our goal is to minimize the loss value. So, a value close to 0 says that the classifier is very good at predicting on the dataset.

Logg Loss requires that your classifier outputs probability for each possible class, instead of just the most likely one. Ideal classifier assigns a probability equal to 1 for the correct class and 0 for all else.

Just as with Linear Regression, we can use Gradient Descent to find the optimal parameters for our model. How can we do it with *scikit-learn*?

[84] http://wiki.fast.ai/index.php/Log_Loss

Example

The LogisticRegression[85] from scikit-learn allows you to do multiclass classification. It also applies $l2$ regularization by default. Let's use it to predict car model origin:

```
from sklearn.linear_model import LogisticRegression

X, y = create_classification_dataset(auto_df)

clf = LogisticRegression(solver="lbfgs")
eval_classifier(clf, X, y)
```

```
0.787948717948718
```

We got about ~79% accuracy, which is quite good, considering how simple the model is.

When To Use It?

Logistic Regression should be your first choice when solving a new classification problem.

Pros:

- Easy to understand and interpret
- Easy to configure (small number of hyperparameters)
- Outputs the likelihood for each class

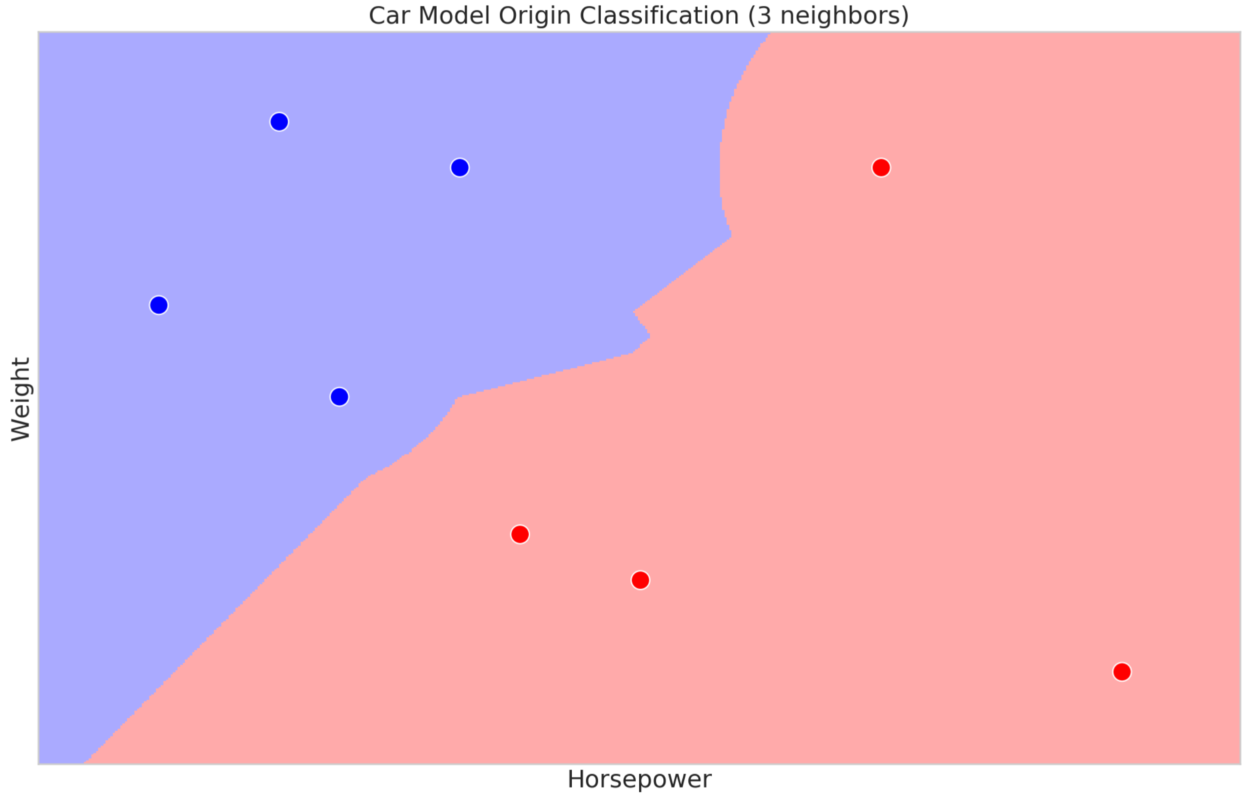
Cons:

- Requires data scaling
- Assumes linear relationship in the data
- Sensitive to outliers

k-Nearest Neighbors

During training, this algorithm stores the data in some sort of efficient data structure (like k-d tree[86]), so it is available for later. Predictions are made by finding k (hence the name) similar training examples and returning the most common label (in case of classification) or avering label values (in case of regression). How do we measure similarity?

[85] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
[86] https://en.wikipedia.org/wiki/K-d_tree



Measuring the similarity of two data points is most commonly done by measuring the distance between them. Some of the most popular distance measures are Euclidean Distance[87]:

$$\text{Eucleadian Distance}(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

measures the straight-line distance between two points in Euclidean space

and Cosine Similarity[88]:

$$\text{Cosine Similarity}(a, b) = \frac{\sum\limits_{i=1}^{n} a_i b_i}{\sqrt{\sum\limits_{i=1}^{n} a_i^2}\sqrt{\sum\limits_{i=1}^{n} b_i^2}}$$

which measures how similar the directions of two vectors are.

You might think that normalizing features is really important for this algorithm, and you'll be right!

[87] https://en.wikipedia.org/wiki/Euclidean_distance
[88] https://en.wikipedia.org/wiki/Cosine_similarity

Example

k-Nearest Neighbors (KNN) can be used for classification and regression tasks. KNeighborsClassifier[89] offers a nice set of options for parameters like - number of neighbors and the type of metric to use. Let's look at an example:

```python
from sklearn.neighbors import KNeighborsClassifier

X, y = create_classification_dataset(auto_df)

clf = KNeighborsClassifier(n_neighbors=24)
eval_classifier(clf, X, y)
```

```
0.8008333333333335
```

How can you find good values for k (number of neighbors)? Usually, you just try a lot of different values.

When To Use It?

This algorithm might have very good performance when compared to Linear Regression. It works quite well on smallish datasets with not that many features.

Pros:

- Easy to understand and reason about
- Trains instantly (all of the work is done when predicting data)
- Makes no assumption about input data distributions
- Automatically adjusts predictions as new data comes in

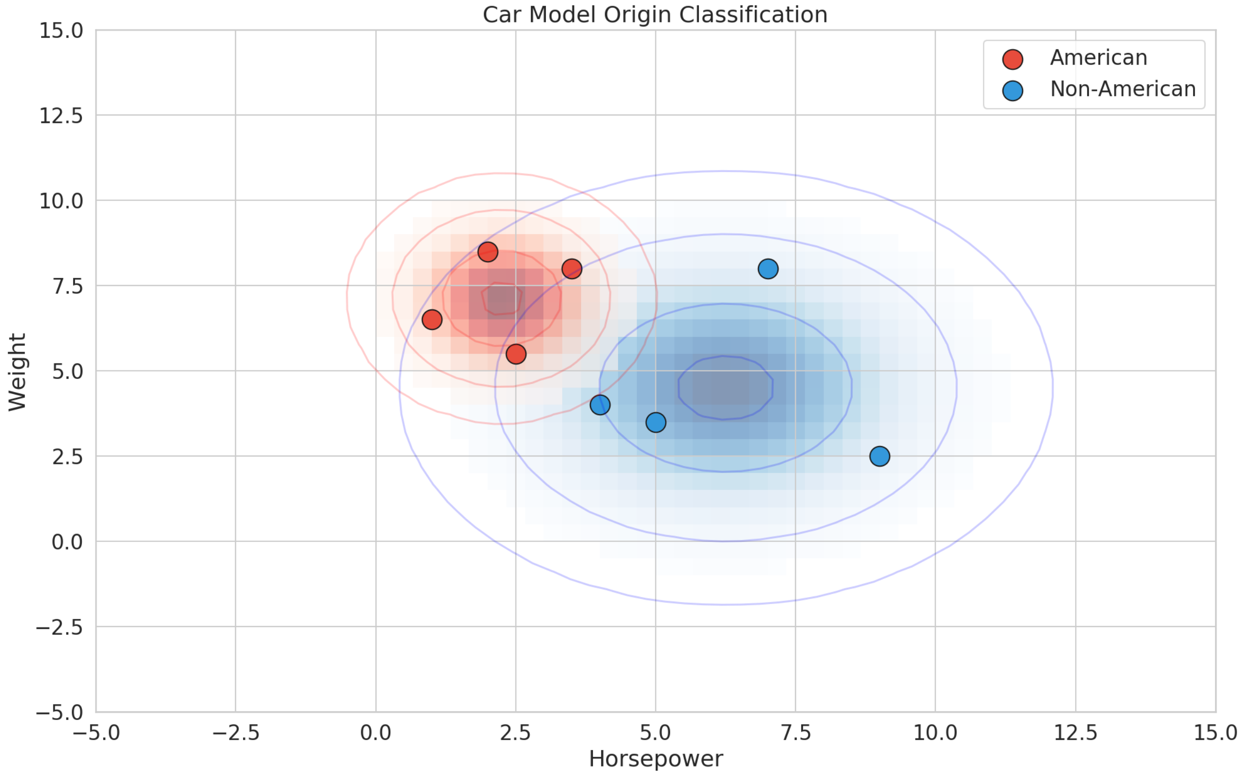
Cons:

- Need to scale numerical features (depends on distance measurements)
- Slow inference time (all of the work is done when predicting data)
- Sensitive to imbalanced datasets - values occuring more often will bias the results. You can use resampling techniques for this issue.
- High dimensional features may produce closeness to many data points. You can apply dimensionality reduction techniques for this issue.

[89] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Naive Bayes

Naive Bayes[90] algorithms calculate the likelihood of each class is correct. They apply Bayes' theorem to classification problems. That is, with a strong (and often unrealistic) assumption of independence between the features.



Bayes Theorem

Bayes theorem[91] gives the following relationship between the labels and features:

$$P(y \mid \mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n}) = \frac{P(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n} \mid y) P(y)}{P(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n})}$$

Using the independence assumption we get:

$$P(y \mid \mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n}) = \frac{P(y) \prod_{i=1}^{n} P(x_i|y)}{P(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n})}$$

$P(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n})$ is a normalizing term (constant). We can drop it, since we're interested in the most probable hypothesis, and use the following classification rule:

[90] https://en.wikipedia.org/wiki/Naive_Bayes_classifier
[91] https://en.wikipedia.org/wiki/Bayes%27_theorem

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

Example

Scikit-learn implements multiple Naive Bayes classifiers. We're going to use GaussianNB[92] which assumes Gaussian distribution of the data:

```
from sklearn.naive_bayes import GaussianNB

X, y = create_classification_dataset(auto_df)

clf = GaussianNB()
eval_classifier(clf, X, y)
```

```
0.7597435897435898
```

When To Use It?

Naive Bayes classifiers are a very good choice for building baseline models that have a probabilistic interpretation of its outputs. Training and prediction speed are very good on large amounts of data.

Pros:

- Fast training and inference performance
- Can be easily interpreted due to probabilistic predictions
- Easy to tune (a few hyperparameters)
- No feature scaling required
- Can handle imbalanced datasets - with Complement Naive Bayes

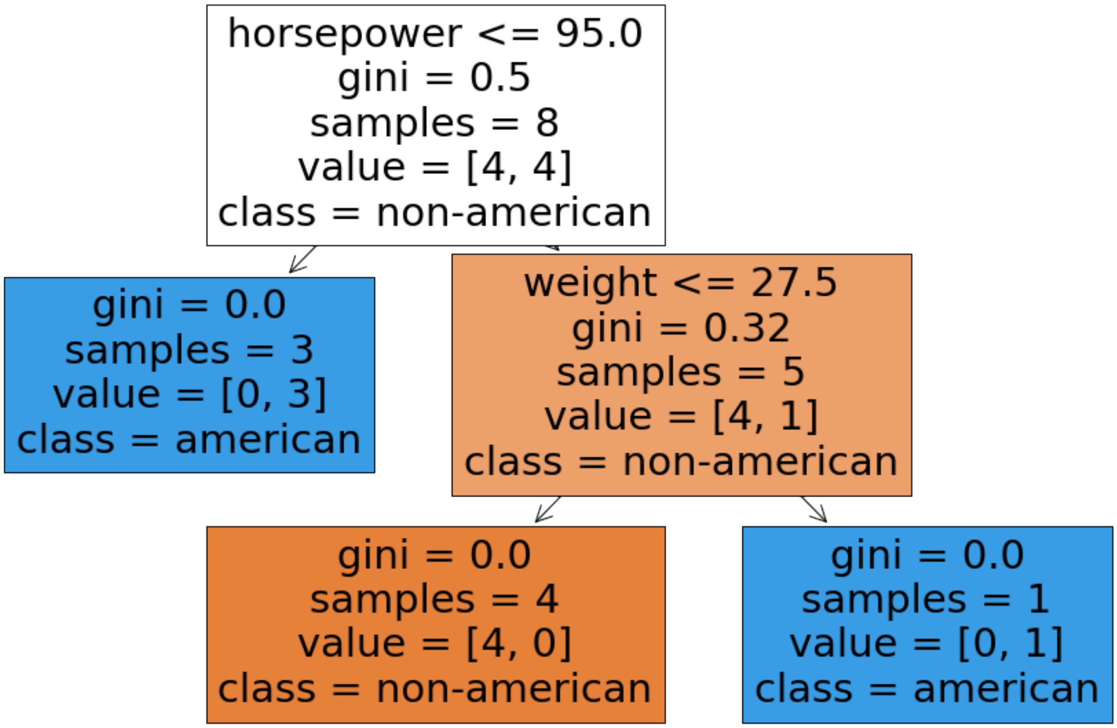
Cons:

- Naive assumption about independence, which is rarely true (duh)
- Performance suffers when multicollinearity[93] is present

[92] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB
[93] https://en.wikipedia.org/wiki/Multicollinearity

Decision Trees

Decision Tree algorithms build (mostly binary) trees using the data to choose split points. At each node, a specific feature is examined and compared to a threshold. We go to the left if the value is below the threshold, else we go right. We get an answer (prediction) of the model when a leaf node is reached.



Example

Scikit-learn offers multiple tree-based algorithms for both regression and classification. Let's look at an example:

```
from sklearn.tree import DecisionTreeRegressor

X, y = create_regression_dataset(auto_df)

reg = DecisionTreeRegressor()
eval_regressor(reg, X, y)
```

```
0.6529188972733717
```

Random Forests

The Random Forest algorithm combines multiple decision trees. Each tree is trained on a random subset of the data and has a low bias (low error on the training data) and high variance (high error on the test data). Aggregating the trees allows you to build a model with low variance.

```python
from sklearn.ensemble import RandomForestRegressor

X, y = create_regression_dataset(auto_df)

reg = RandomForestRegressor(n_estimators=50)
eval_regressor(reg, X, y)
```

```
0.3976871715935767
```

Note the error difference between a single Decision Tree and a Random Forest with 50 weak Decision Trees.

Boosting

This method builds multiple decision trees iteratively. Each new tree tries to fix the errors made by the previous one. At each step, the error between the predicted and actual data is added to the loss and then minimized at the next step.

```python
from sklearn.ensemble import GradientBoostingRegressor

X, y = create_regression_dataset(auto_df)

reg = GradientBoostingRegressor(n_estimators=100)
eval_regressor(reg, X, y)
```

```
0.37605497373246266
```

Now go to Kaggle[94] and check how many competitions are won by using this method.

[94] https://www.kaggle.com/

When To Use It?

In practice, you'll never be using a single Decision Tree. Ensemble methods (multiple decision trees) are the way to go. Overall, boosting can give you the best possible results (especially when using libraries like LightGBM[95]). But you have a lot of hyperparameters to tune. It might take a lot of time and experience to develop really good models.

Pros:

- Easy to interpret and visualize (white box models)
- Can handle numerical and categorical data
- No complex data preprocessing - no normalization or missing data imputation is need
- Fast prediction speed - O(log n)
- Can be used in ensembles to prevent overfitting and increase accuracy
- Perform very well on both regression and classification tasks
- Show feature importances

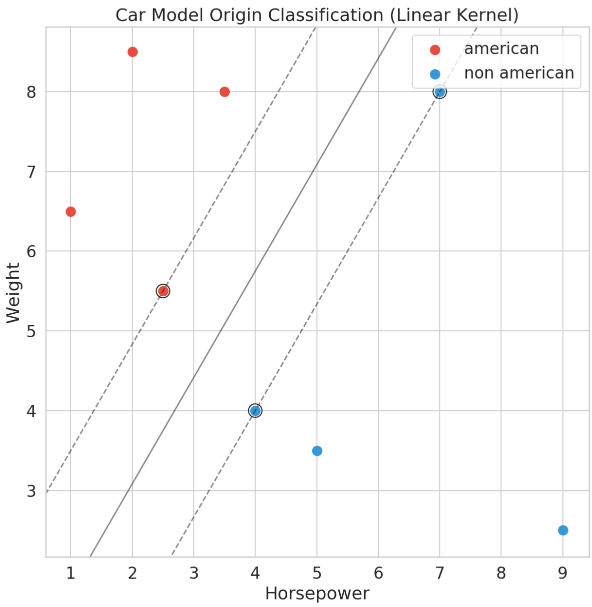
Cons:

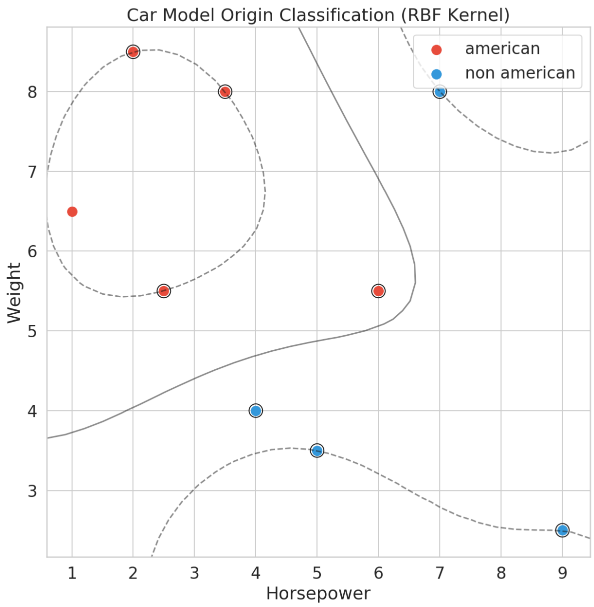
- Do not work well with imbalanced datasets - fixed by balancing or providing class weights
- Easy to overfit - you can build very deep trees that memorize every feature value - fixed by limiting tree depth
- Must be used in ensembles to get good results in practice
- Sensitive to data changes (small variation can build entirely different tree) - fixed using ensembles

Support Vector Machines (SVM)

SVM models try to build hyperplanes (n-dimensional lines) that best separate the data. Hyperplanes are created such that there is a maximum distance between the closest example of each class.

[95] https://github.com/microsoft/LightGBM





Hard-margin

Hard-margin SVMs[96] work when the data is linearly separable. We want to minimize the margin between the support vectors $\|w\|$ (the closest data points to the separating hyperplane). We have:

$$\min \frac{1}{2}\|w\|^2$$

satisfying the constraint:

$$y_i(wx_i - b) - 1 \geq 0, i = 1, \ldots, n$$

What about data points that cannot be linearly separated?

Soft-margin

In practice, the expectation that the data is linearly separable is unrealistic. We can cut some slack to our SVM and introduce a constant C. It determines the tradeoff between increasing the decision boundary and placing each data point on the correct side of the decision boundary.

We want to minimize the following function:

$$C\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i(wx_i - b))$$

[96] https://en.wikipedia.org/wiki/Support-vector_machine#Hard-margin

Choosing the correct C is done experimentally. You can look at this parameter as a way to control the bias-variance tradeoff for your model.

Example

Using SVMs on regression problems can be done using the SVR[97] model:

```python
from sklearn.svm import SVR

X, y = create_regression_dataset(auto_df)

reg = SVR(gamma="auto", kernel="rbf", C=4.5)
eval_regressor(reg, X, y)
```

```
0.32820308689067834
```

When To Use It?

Support Vector Machines can give you great performance but need careful tuning. You can solve non-linearly separable problems with a proper choice of a kernel function.

Pros:

- Can provide very good results used for regression and classification
- Can learn non-linear boundaries (see the kernel trick[98])
- Robust to overfitting in higher dimensional space

Cons:

- Large number of hyperparameters
- Data must be scaled
- Data must be balanced
- Sensitive to outliers - can be mitigated by using soft-margin

[97] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

[98] https://en.wikipedia.org/wiki/Kernel_method#Mathematics:_the_kernel_trick

Conclusion

You covered some of the most used Machine Learning algorithms. But you've just scratched the surface. Devil is in the details, and those algorithms have a lot of details surrounding them.

You learned about:

- Linear Regression
- Logistic Regression
- k-Nearest Neighbors
- Naive Bayes
- Decision Trees
- Support Vector Machines

I find it fascinating that there are no clear winners when it comes to having an all-around best algorithm. Your project/problem will inevitably require some careful experimentation and planning. Enjoy the process :)

Run the complete notebook in your browser[99]

The complete project on GitHub[100]

References

- Machine Learning Notation[101]
- Making Sense of Logarithmic Loss[102]
- In Depth: Naive Bayes Classification[103]

[99] https://colab.research.google.com/drive/1-_wQbYW-KqDNMkT9iZ-d2KWVHyR6d0nn
[100] https://github.com/curiousily/Deep-Learning-For-Hackers
[101] https://nthu-datalab.github.io/ml/slides/Notation.pdf
[102] https://datawookie.netlify.com/blog/2015/12/making-sense-of-logarithmic-loss/
[103] https://jakevdp.github.io/PythonDataScienceHandbook/05.05-naive-bayes.html

Data Preprocessing

> TL;DR Learn how to do feature scaling, handle categorical data and do feature engineering with Pandas and Scikit-learn in Python. Use your skills to preprocess a housing dataset and build a model to predict prices.

I know, data preprocessing might not sound cool. You might just want to train Deep Neural Networks (or your favorite models). I am here to shatter your dreams, you'll most likely spend a lot more time on data preprocessing and exploration[104] than any other step of your Machine Learning workflow.

Since this step is so early in the process, screwing up here will lead to useless models. **Garbage data in**, **garbage predictions out**. A requirement for reaching your model's full potential is proper cleaning, wrangling and analysis of the data.

This guide will introduce you to the most common and useful methods to preprocess your data. We're going to look at three general techniques:

- Feature Scaling
- Handling Categorical Data
- Feature Engineering

Finally, we're going to apply what we've learned on a real dataset and try to predict Melbourne housing prices. We're going to compare the performance of a model with and without data preprocessing. How improtant data preparation really is?

Run the complete notebook in your browser[105]

The complete project on GitHub[106]

Feature Scaling

Feature scaling[107] refers to the process of changing the range (normalization) of numerical features. There are different methods to do feature scaling. But first, why do you need to do it?

When Machine Learning algorithms measure distances between data points, the results may be dominated by the magnitude (scale) of the features instead of their values. Scaling the features to a

[104] https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/#305db5686f63

[105] https://colab.research.google.com/drive/1c61XEZ7MHKFDcBOX87Wx1SNrtNYAF6Zt

[106] https://github.com/curiously/Deep-Learning-For-Hackers

[107] https://en.wikipedia.org/wiki/Feature_scaling

similar range can fix the problem. Gradient Descent[108] can converge faster[109] when feature scaling is applied.

> Use feature scaling when your algorithm calculates distances or is trained with Gradient Descent

How can we do feature scaling? Scikit-learn[110] offers a couple of methods. We'll use the following synthetic data to compare them:

```
data = pd.DataFrame({
    'Normal': np.random.normal(100, 50, 1000),
    'Exponential': np.random.exponential(25, 1000),
    'Uniform': np.random.uniform(-150, -50, 1000)
})
```

Min-Max Normalization

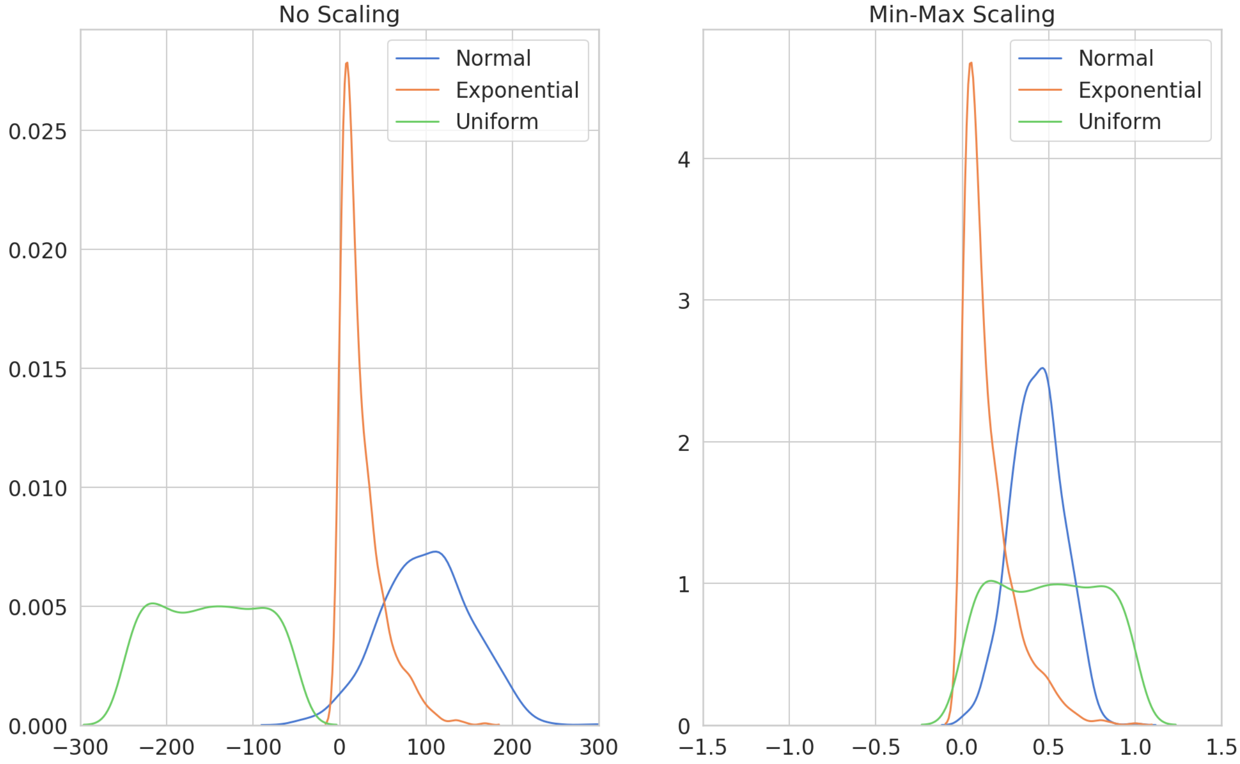
One of the simplest and most widely used approaches is to scale each feature in the [0, 1] range. The scaled value is given by:

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

MinMaxScaler[111] allows you to select the rescale range with the `feature_range` parameter:

```
from sklearn.preprocessing import MinMaxScaler

min_max_scaled = MinMaxScaler(feature_range=(0, 1)).fit_transform(data)
```

[108] https://en.wikipedia.org/wiki/Gradient_descent
[109] https://arxiv.org/abs/1502.03167
[110] https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing
[111] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html



The scaled distributions do not overlap as much and their shape remains the same (except for the Normal).

This method preserves the shape of the original distribution and is sensitive to outliers.

Standardization

This method rescales a feature removing the mean and divides by standard deviation. It produces a distribution centered at 0 with a standard deviation of 1. Some Machine Learning algorithms (SVMs) assume features are in this range.

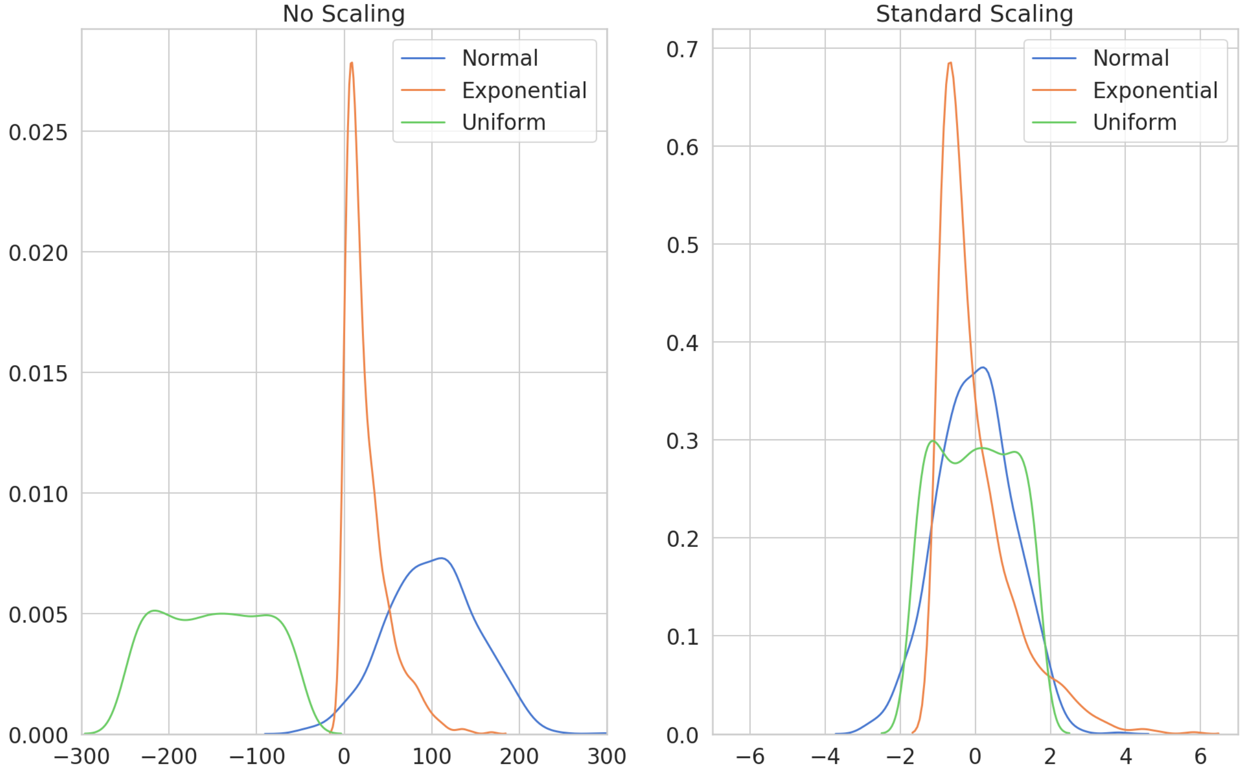
It is defined by:

$$x' = \frac{x - \text{mean(x)}}{\text{stdev(x)}}$$

You can use the StandarScaler[112] like this:

[112] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

```
from sklearn.preprocessing import StandardScaler

stand_scaled = StandardScaler().fit_transform(data)
```



The resulting distributions overlap heavily. Also, their shape is much narrower.

This method "makes" a feature normally distributed. With outliers, your data will be scaled to a small interval.

Robust Scaling

This method is very similar to the Min-Max approach. Each feature is scaled with:

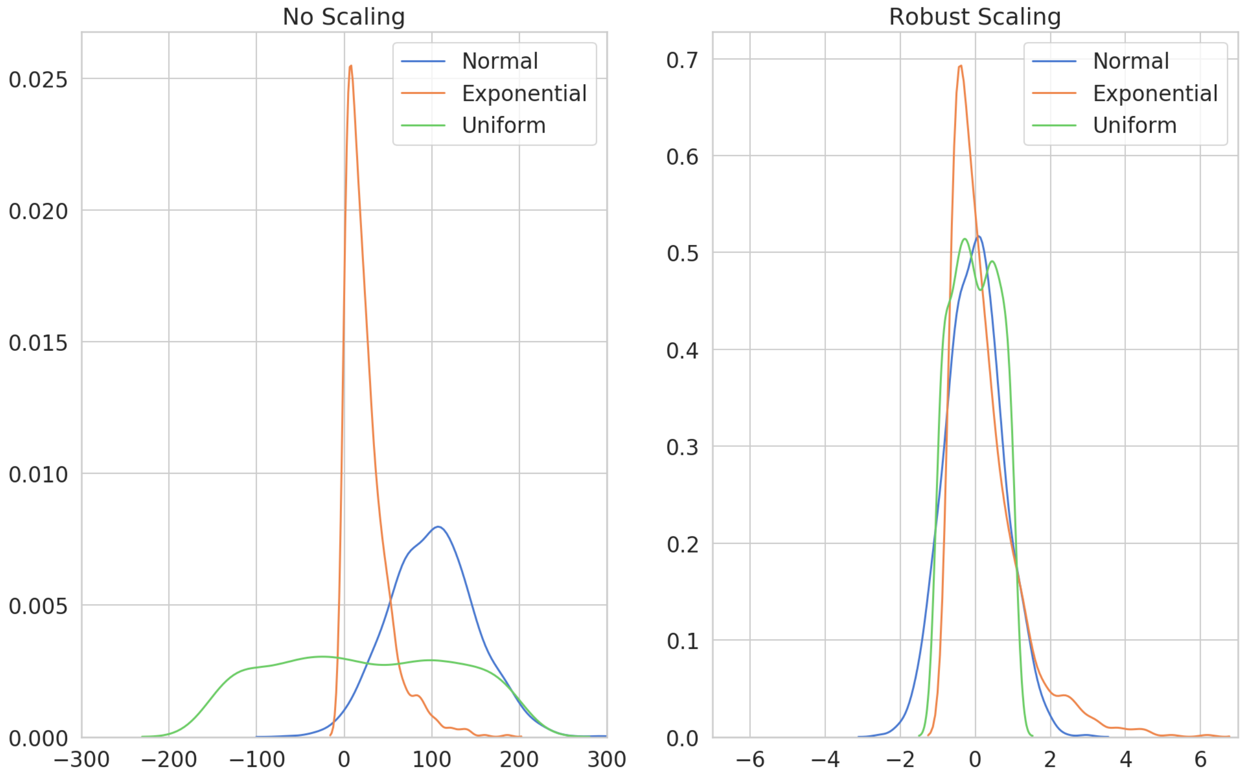
$$X = \frac{x - Q_1(x)}{Q_3(x) - Q_1(x)}$$

where Q are quartiles. The Interquartile range[113] makes this method robust to outliers (hence the name).

Let's use the RobustScaler[114] on our data:

[113] https://en.wikipedia.org/wiki/Interquartile_range
[114] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html

```python
from sklearn.preprocessing import RobustScaler

robust_scaled = RobustScaler().fit_transform(data)
```

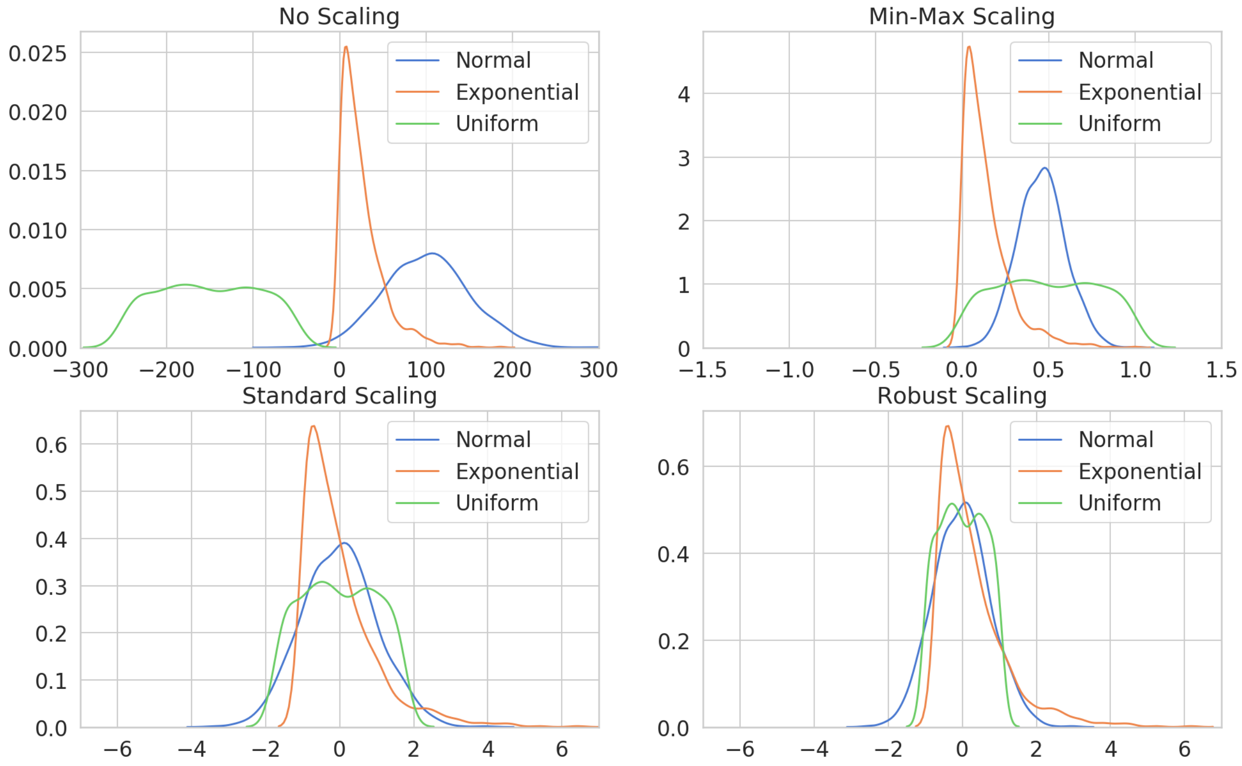


All distributions have most of their densities around 0 and a shape that is more or less the same.

Use this method when you have outliers and want to minimize their influence.

Scaling Methods Overview

Here’s an overview of the scaled distributions compared to the non-scaled version:



Handling Categorical Data

Categorical variables (also known as nominal[115]) are a set of enumerable values. They cannot be numerically organized or ranked. How can we use them in our Machine Learning algorithms?

Some algorithms, like decision trees, will work fine without any categorical data preprocessing. Unfortunatelly, that is the exception rather than the rule.

How can we encode the following property types?

```
property_type =\
    np.array(['House', 'Unit', 'Townhouse', 'House', 'Unit'])
    .reshape(-1, 1)
```

Integer Encoding

Most Machine Learning algorithms require numeric-only data. One simple way to achieve that is to assing an unique value to each category.

We can use the OrdinalEncoder[116] for that:

[115] https://en.wikipedia.org/wiki/Nominal_category
[116] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html

```
from sklearn.preprocessing import OrdinalEncoder

enc = OrdinalEncoder().fit(property_type)
labels = enc.transform(property_type)
labels.flatten()
```

```
array([0., 2., 1., 0., 2.])
```

You can obtain the string representation of the categories like so:

```
enc.inverse_transform(one_hots).flatten()
```

```
array(['House', 'Unit', 'Townhouse', 'House', 'Unit'], dtype='<U9')
```

One-Hot Encoding

Unfortunatelly, the simple integer encoding makes the assumption that the categories can be ordered (ranked).

Sometimes, that assumption might be correct. When it is not, you can use one-hot encoding[117]:

```
from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder(sparse=False).fit(property_type)
one_hots = enc.transform(property_type)
one_hots
```

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

Basically, one-hot encoding creates a vector of zeros for each row in our data with a one at the index (place) of the category.

This solves the ordering/ranking issue but introduces another one. Each categorical feature creates k (number of unique categories) new columns in our dataset, which are mostly zeros.

[117] https://en.wikipedia.org/wiki/One-hot

How Many Categories are Too Many for One-Hot Encoding?

With a vast amounts (number of rows) of data you might be able to get away with encoding lots of categorical features with a lot of categories.

Here are some ways to tackle the problem, when that is not possible:

- Drop insignificant features before encoding
- Drop columns with mostly zeros after encoding
- Create aggregate (larger) categories and one-hot encode them
- Encode them as integers and test your model performance

Adding New Features

Feature engineering refers to the process of augmenting your data (usually by adding features) using your (human) knowledge. This often improves the performance of Machine Learning algorithms.

Deep Learning[118] has changed the feature engineering game when it comes to text and image data. Those algorithms learn intermediate representations of the data. In a way, they do automatic feature engineering.

When it comes to structured data (think data you get with SQL queries from your database), feature engineering might give you a lot of performance improvement.

How can we improve our datasets?

Turn Numbers into Categories

You already know how to convert categorical data to numbers. You can also turn ranges (bins) of numbers into categories. Let's see an example:

```
n_rooms = np.array([1, 2, 1, 4, 6, 7, 12, 20])
```

We'll turn the number of rooms into three categories - small, medium and large:

```
pd.cut(n_rooms, bins=[0, 3, 8, 100], labels=["small", "medium", "large"])
```

```
[small, small, small, medium, medium, medium, large, large]
Categories (3, object): [small < medium < large]
```

The `cut()`[119] function from Pandas gives you a way to turn numbers into categories by specifying ranges and labels. Of course, you can use one-hot encoding on the new categories.

[118] https://en.wikipedia.org/wiki/Deep_learning
[119] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html

Extract Features from Dates

Dates in computers are represented as milliseconds since the Unix epoch[120] - 00:00:00 UTC on 1 January 1970. You can use the raw numbers or extract some information from the dates. How can we do this with Pandas?

```
dates = pd.Series(["1/04/2017", "2/04/2017", "3/04/2017"])
```

You can convert the string formatted dates into date objects with `to_datetime()`[121]. This function works really well on a variety of formats. Let's convert our dates:

```
pd_dates = pd.to_datetime(dates)
```

One important feature we can get from the date values is the day of the week:

```
pd_dates.dt.dayofweek
```

```
0    2
1    5
2    5
dtype: int64
```

There you go, even more categorical data :)

Predicting Melbourne Housing Prices

Let's use our new skills to do some data preprocessing on a real-world data. We'll use the Melbourne Housing Market dataset available on Kaggle[122].

The Data

Here's the description of the data:

> This data was scraped from publicly available results posted every week from Domain.com.au, I've cleaned it as best I can, now it's up to you to make data analysis magic. The dataset includes Address, Type of Real estate, Suburb, Method of Selling, Rooms, Price, Real Estate Agent, Date of Sale and distance from C.B.D.

Our task is to predict the sale price of the property based on a set of features. Let's get the data using gdown:

[120] https://en.wikipedia.org/wiki/Unix_time

[121] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_datetime.html

[122] https://www.kaggle.com/anthonypino/melbourne-housing-market

```
!gdown --id 1bIa7HOtpakl1Qzn6pmKCMAzrWjM08mfI --output melbourne_housing.csv
```

And load it into a Pandas dataframe:

```
df = pd.read_csv('melbourne_housing.csv')
df.shape
```

```
(34857, 21)
```

We have almost 35k rows and 21 columns. Here are the features:

- Suburb
- Address
- Rooms
- Type - br - bedroom(s); h - house,cottage,villa, semi,terrace; u - unit, duplex; t - townhouse; dev site - development site; o res - other residential.
- Price - price in Australian dollars
- Method - S - property sold; SP - property sold prior; PI - property passed in; PN - sold prior not disclosed; SN - sold not disclosed; NB - no bid; VB - vendor bid; W - withdrawn prior to auction; SA - sold after auction; SS - sold after auction price not disclosed. N/A - price or highest bid not available.
- SellerG
- Date - date sold
- Distance
- Postcode
- Bedroom2
- Bathroom
- Car - number of carspots
- Landsize - land size in meters
- BuildingArea - building size in meters
- YearBuilt
- CouncilArea
- Lattitude
- Longtitude
- Regionname
- Propertycount - number of properties in the suburb

Let's check for missing values:

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

```
BuildingArea     21115
YearBuilt        19306
Landsize         11810
Car               8728
Bathroom          8226
Bedroom2          8217
Longtitude        7976
Lattitude         7976
Price             7610
Propertycount        3
Regionname           3
CouncilArea          3
Postcode             1
Distance             1
dtype: int64
```

We have a lot of those. For the purpose of this guide, we're just going to drop all rows that contain missing values:

```
df = df.dropna()
```

Predicting without Preprocessing

Let's use the "raw" features to train a model and evaluate its performance. First, let's split the data into training and test sets:

```
X = df[[
  'Rooms', 'Distance', 'Propertycount',
  'Postcode', 'Lattitude', 'Longtitude'
]]
y = np.log1p(df.Price.values)

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

We'll use the GradientBoostingRegressor[123] and train it on our data:

[123] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

```python
from sklearn.ensemble import GradientBoostingRegressor

forest = GradientBoostingRegressor(
  learning_rate=0.3, n_estimators=150, random_state=RANDOM_SEED
).fit(X_train, y_train)

forest.score(X_test, y_test)
```

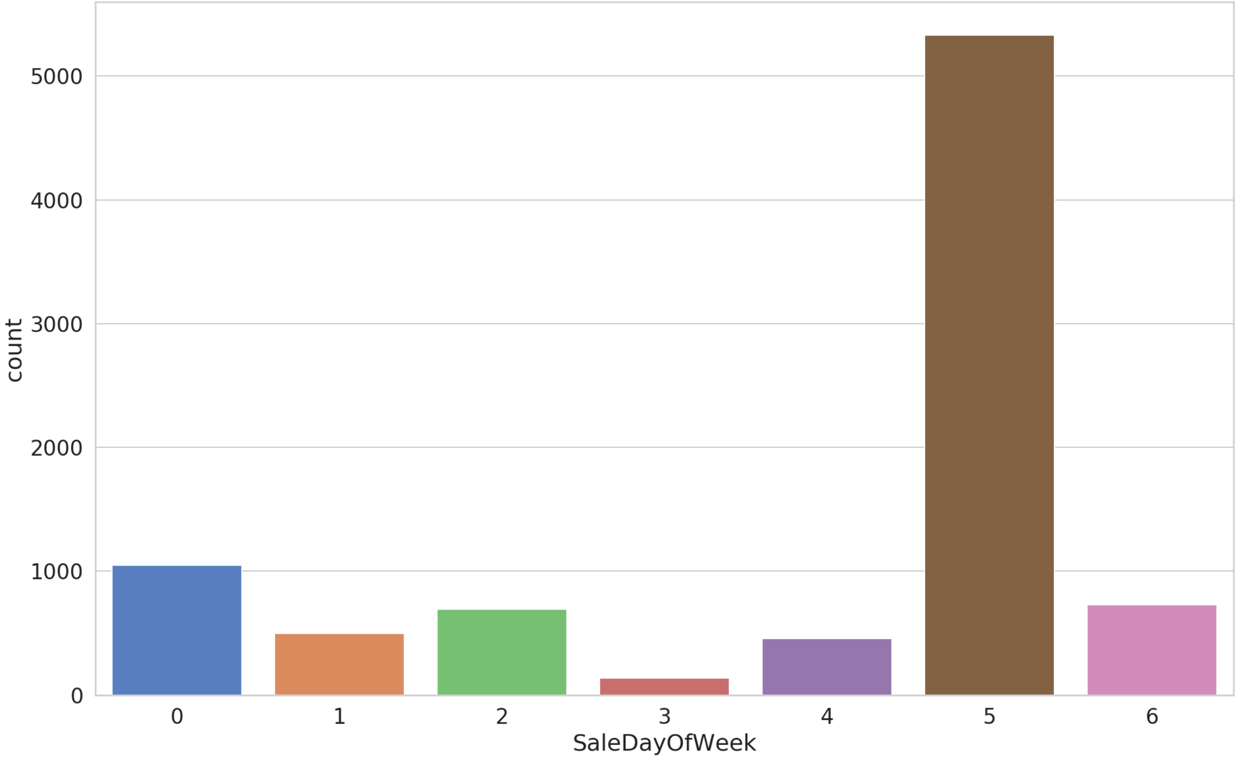
```
0.7668970798114849
```

Good, now you have a baseline R^2 score on the raw data.

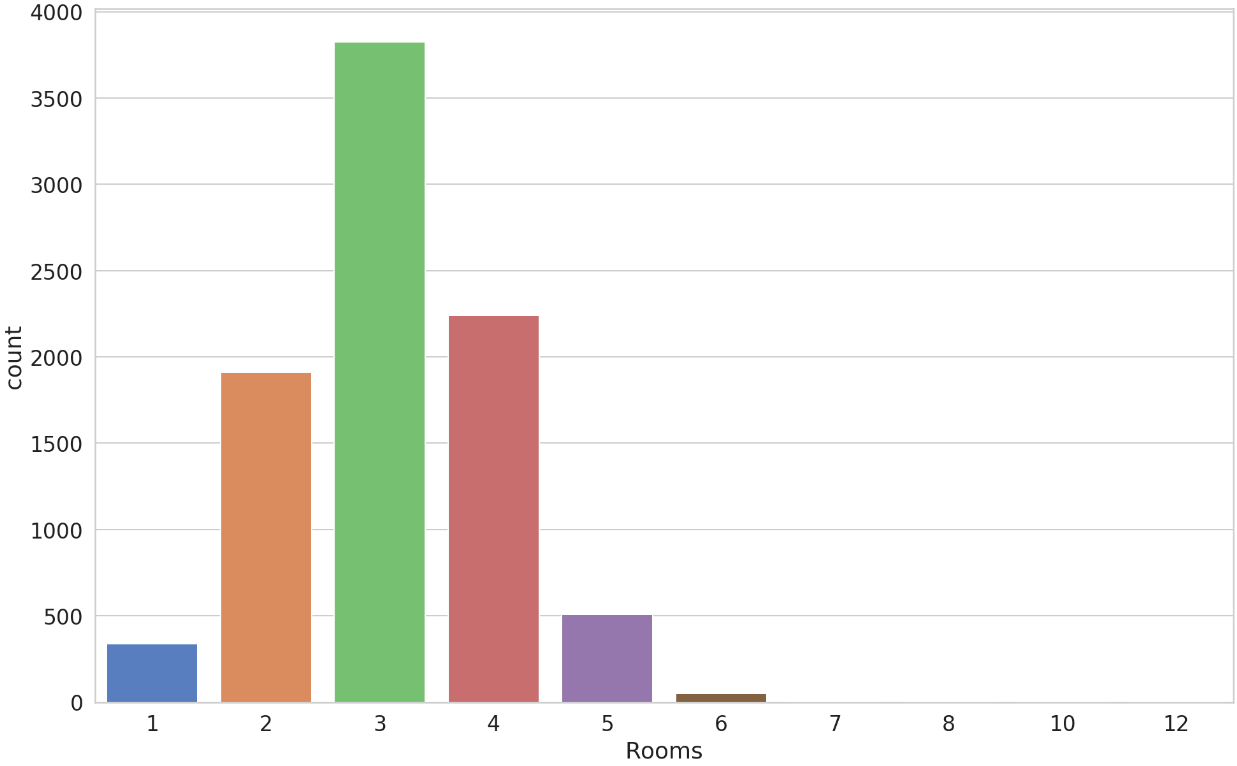
Preprocessing

Let's start with something simple - extract the sale day of the week. We'll add that to our dataset. You already know how to do this:

```python
df['Date'] = pd.to_datetime(df.Date)
df['SaleDayOfWeek'] = df.Date.dt.dayofweek
```



Saturday looks like a really important day for selling properties. Let's have a look at the number of rooms:



We can use the binning technique to create categories from the rooms:

```
df['Size'] = pd.cut(
  df.Rooms,
  bins=[0, 2, 4, 100],
  labels=["Small", "Medium", "Large"]
)
```

Next, let's drop some of the columns we're not going to use:

```
df = df.drop(['Address', 'Date'], axis=1)
```

Let's create the training and test datasets:

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

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

The `make_column_transformer()`[124] allows you to build an uber transformer™ composed of multiple transformers. Let's use it on our data:

```
from sklearn.compose import make_column_transformer

transformer = make_column_transformer(
    (RobustScaler(),
      [
        'Distance', 'Propertycount', 'Postcode',
        'Lattitude', 'Longtitude', 'Rooms'
      ]),
    (OneHotEncoder(handle_unknown="ignore"),
      ['Size', 'SaleDayOfWeek', 'Type', 'Method', 'Regionname']),
    (OrdinalEncoder(
        categories=[
          X.CouncilArea.unique(),
          X.SellerG.unique(),
          X.Suburb.unique()],
        dtype=np.int32
      ), ['CouncilArea', 'SellerG', 'Suburb']
    ),
)
```

We'll let the transformer learn only from the training data. That is vital since we don't want our RobustScaler to leak information from the test set via the rescaled mean and variance.

Always: split the data into training and test set, then apply preprocessing

```
transformer.fit(X_train)

X_train = transformer.transform(X_train)
X_test = transformer.transform(X_test)
```

Will your model perform better with the preprocessed data?

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

Predicting with Preprocessing

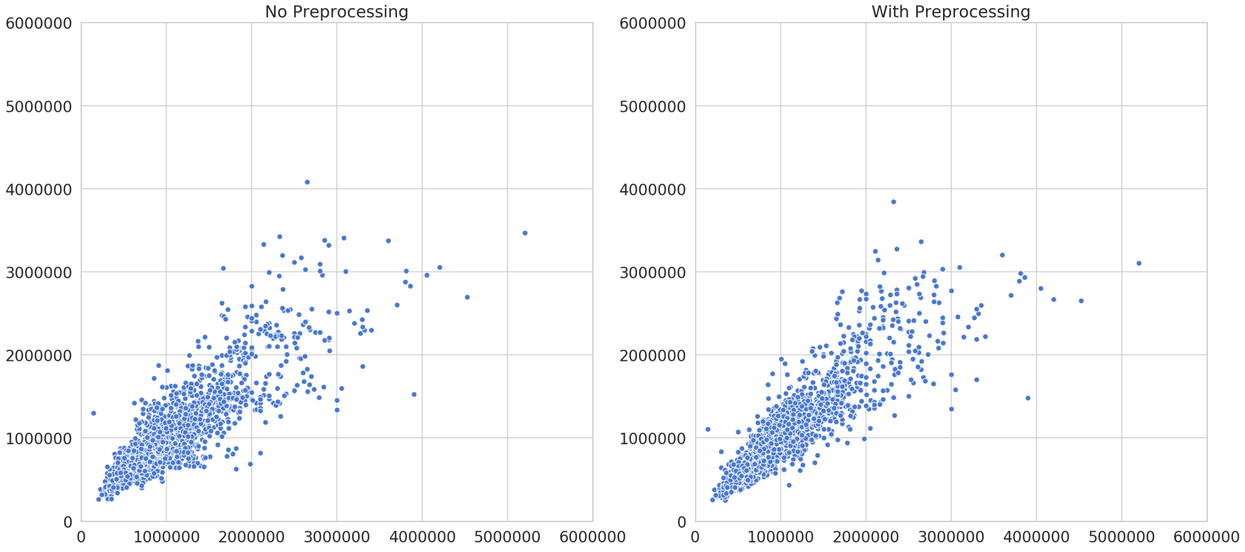
We'll reuse the same model and train it on the preprocessed dataset:

```
forest = GradientBoostingRegressor(
  learning_rate=0.3,
  n_estimators=150,
  random_state=RANDOM_SEED
).fit(X_train, y_train)
forest.score(X_test, y_test)
```

```
0.8393772235062138
```

Considering that our baseline model was doing pretty well, you might be surprised by the improvement. It is definitely something.

Here's a comparison of the predictions:



You can see that the predictions are looking much better (better predictions lie on the diagonal). Can you come up with more features/preprocessing to improve the R^2 score?

Conclusion

You've learned about some of the useful data preprocessing techniques. You've also applied what you've learned to a real-world dataset for predicting Melbourne Housing prices. Here's an overview of the methods used:

- Feature Scaling
- Handling Categorical Data
- Feature Engineering

Do you use any other techniques to prepare your data?

Run the complete notebook in your browser[125]

The complete project on GitHub[126]

References

- Gradient descent in practice I: Feature Scaling[127]
- Compare the effect of different scalers on data with outliers[128]
- Feature Scaling[129]
- Melbourne Housing Market[130]

[125] https://colab.research.google.com/drive/1c61XEZ7MHKFDcBOX87Wx1SNrtNYAF6Zt

[126] https://github.com/curiousily/Deep-Learning-For-Hackers

[127] https://www.youtube.com/watch?v=e1nTgoDI_m8

[128] https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html#sphx-glr-auto-examples-preprocessing-plot-all-scaling-py

[129] https://jovianlin.io/feature-scaling/

[130] https://www.kaggle.com/anthonypino/melbourne-housing-market

Handling Imbalanced Datasets

> TL;DR Learn how to handle imbalanced data using TensorFlow 2, Keras and scikit-learn

Datasets in the wild will throw a variety of problems towards you. What are the most common ones?

The data might have too few examples, too large to fit into the RAM, multiple missing values, do not contain enough predictive power to make correct predictions, and it can imbalanced.

In this guide, we'll try out different approaches to solving the imbalance issue for classification tasks. That isn't the only issue on our hands. Our dataset is real, and we'll have to deal with multiple problems - imputing missing data and handling categorical features.

Before getting any deeper, you might want to consider far simpler solutions to the imbalanced dataset problem:

- Collect more data - This might seem like a no brainer, but it is often overlooked. Can you write some more queries and extract data from your database? Do you need a few more hours for more customer data? More data can balance your dataset or might make it even more imbalanced. Either way, you want a more complete picture of the data.
- Use Tree based models - Tree-based models tend to perform better on imbalanced datasets. Essentially, they build hierarchies based on split/decision points, which might better separate the classes.

Here's what you'll learn:

- Impute missing data
- Handle categorical features
- Use the right metrics for classification tasks
- Set per class weights in Keras when training a model
- Use resampling techniques to balance the dataset

Run the complete code in your browser[131]

Data

Naturally, our data should be imbalanced. Kaggle has the perfect one for us - Porto Seguro's Safe Driver Prediction[132]. The object is to predict whether a driver will file an insurance claim. How many drivers do that?

[131] https://colab.research.google.com/drive/1lZvXQxaO4mOT3-zImEkb0boDctju0cLw

[132] https://www.kaggle.com/c/porto-seguro-safe-driver-prediction

Setup

Let's start with installing TensorFlow and setting up the environment:

```
!pip install tensorflow-gpu
!pip install gdown
```

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import pandas as pd

RANDOM_SEED = 42

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

We'll use gdown[133] to get the data from Google Drive:

```
!gdown --id 18gwvNkMs6t0jL0APl9iWPrhr5GVg082S --output insurance_claim_prediction.csv
```

Exploration

Let's load the data in Pandas[134] and have a look:

```
df = pd.read_csv('insurance_claim_prediction.csv')
print(df.shape)j
```

```
(595212, 59)
```

Loads of data. What features does it have?

```
print(df.columns)
```

[133] https://pypi.org/project/gdown/
[134] https://pandas.pydata.org/

```
Index(['id', 'target', 'ps_ind_01', 'ps_ind_02_cat', 'ps_ind_03',
       'ps_ind_04_cat', 'ps_ind_05_cat', 'ps_ind_06_bin', 'ps_ind_07_bin',
       'ps_ind_08_bin', 'ps_ind_09_bin', 'ps_ind_10_bin', 'ps_ind_11_bin',
       'ps_ind_12_bin', 'ps_ind_13_bin', 'ps_ind_14', 'ps_ind_15',
       'ps_ind_16_bin', 'ps_ind_17_bin', 'ps_ind_18_bin', 'ps_reg_01',
       'ps_reg_02', 'ps_reg_03', 'ps_car_01_cat', 'ps_car_02_cat',
       'ps_car_03_cat', 'ps_car_04_cat', 'ps_car_05_cat', 'ps_car_06_cat',
       'ps_car_07_cat', 'ps_car_08_cat', 'ps_car_09_cat', 'ps_car_10_cat',
       'ps_car_11_cat', 'ps_car_11', 'ps_car_12', 'ps_car_13', 'ps_car_14',
       'ps_car_15', 'ps_calc_01', 'ps_calc_02', 'ps_calc_03', 'ps_calc_04',
       'ps_calc_05', 'ps_calc_06', 'ps_calc_07', 'ps_calc_08', 'ps_calc_09',
       'ps_calc_10', 'ps_calc_11', 'ps_calc_12', 'ps_calc_13', 'ps_calc_14',
       'ps_calc_15_bin', 'ps_calc_16_bin', 'ps_calc_17_bin', 'ps_calc_18_bin',
       'ps_calc_19_bin', 'ps_calc_20_bin'],
      dtype='object')
```

Those seem somewhat cryptic, here is the data description:

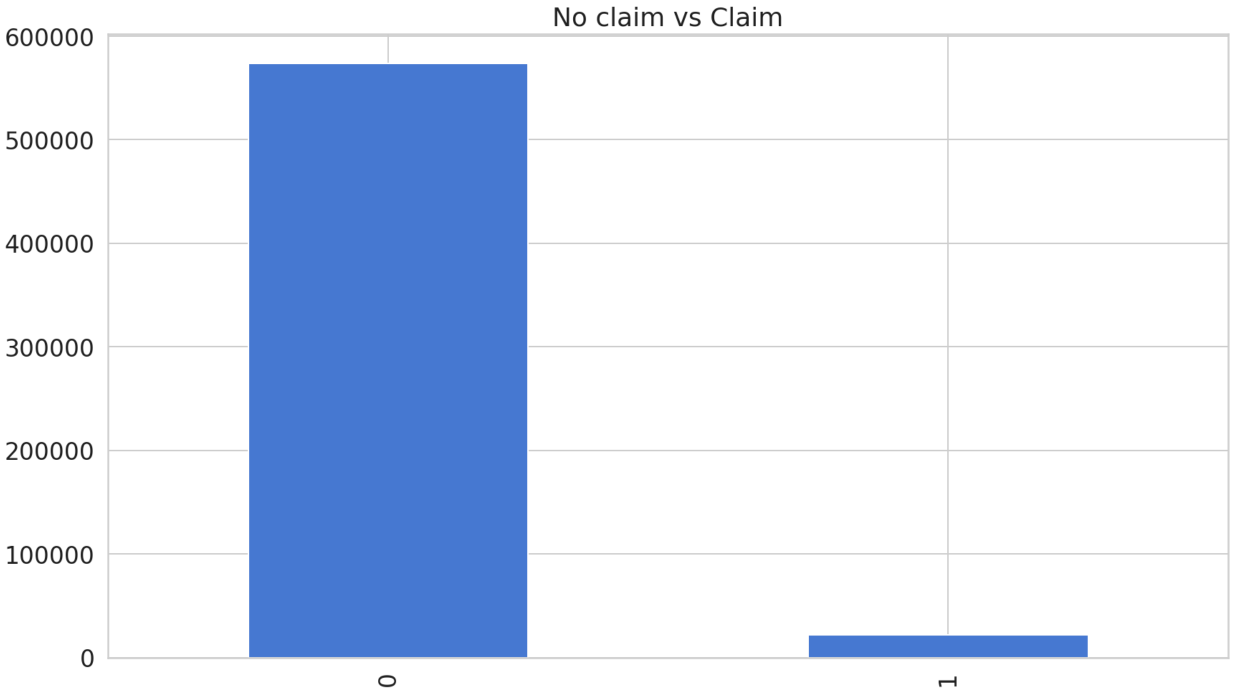
> features that belong to similar groupings are tagged as such in the feature names (e.g., **ind**, **reg**, **car**, **calc**). In addition, feature names include the postfix **bin** to indicate binary features and **cat** to indicate categorical features. Features without these designations are either continuous or ordinal. Values of -1 indicate that the feature was missing from the observation. The **target** columns signifies whether or not a claim was filed for that policy holder.

What is the proportion of each target class?

```
no_claim, claim = df.target.value_counts()
print(f'No claim {no_claim}')
print(f'Claim {claim}')
print(f'Claim proportion {round(percentage(claim, claim + no_claim), 2)}%')
```

```
No claim 573518
Claim 21694
Claim proportion 3.64%
```

Good, we have an imbalanced dataset on our hands. Let's look at a graphical representation of the imbalance:



You got the visual proof right there. But how good of a model can you build using this dataset?

Baseline model

You might've noticed something in the data description. Missing data points have a value of -1. What should we do before training our model?

Data preprocessing

Let's check how many rows/columns contain missing data:

```
row_count = df.shape[0]

for c in df.columns:
  m_count = df[df[c] == -1][c].count()
  if m_count > 0:
    print(f'{c} - {m_count} ({round(percentage(m_count, row_count), 3)}%) rows missi\
ng')
```

```
ps_ind_02_cat - 216 (0.036%) rows missing
ps_ind_04_cat - 83 (0.014%) rows missing
ps_ind_05_cat - 5809 (0.976%) rows missing
ps_reg_03 - 107772 (18.106%) rows missing
ps_car_01_cat - 107 (0.018%) rows missing
ps_car_02_cat - 5 (0.001%) rows missing
ps_car_03_cat - 411231 (69.09%) rows missing
ps_car_05_cat - 266551 (44.783%) rows missing
ps_car_07_cat - 11489 (1.93%) rows missing
ps_car_09_cat - 569 (0.096%) rows missing
ps_car_11 - 5 (0.001%) rows missing
ps_car_12 - 1 (0.0%) rows missing
ps_car_14 - 42620 (7.16%) rows missing
```

Missing data imputation

`ps_car_03_cat`, `ps_car_05_cat` and `ps_reg_03` have too many missing rows for our own comfort. We'll get rid of them. Note that this is not the best strategy but will do in our case.

```
df.drop(
  ["ps_car_03_cat", "ps_car_05_cat", "ps_reg_03"],
  inplace=True,
  axis=1
)
```

What about the other features? We'll use the SimpleImputer from scikit-learn[135] to replace the missing values:

```
from sklearn.impute import SimpleImputer

cat_columns = [
  'ps_ind_02_cat', 'ps_ind_04_cat', 'ps_ind_05_cat',
  'ps_car_01_cat', 'ps_car_02_cat', 'ps_car_07_cat',
  'ps_car_09_cat'
]
num_columns = ['ps_car_11', 'ps_car_12', 'ps_car_14']

mean_imp = SimpleImputer(missing_values=-1, strategy='mean')
cat_imp = SimpleImputer(missing_values=-1, strategy='most_frequent')

```

[135] https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html

```
for c in cat_columns:
  df[c] = cat_imp.fit_transform(df[[c]]).ravel()

for c in num_columns:
  df[c] = mean_imp.fit_transform(df[[c]]).ravel()
```

We use the most frequent value for categorical features. Numerical features are replaced with the mean number of the column.

Categorical features

Pandas `get_dummies()`[136] uses one-hot encoding to represent categorical features. Perfect! Let's use it:

```
df = pd.get_dummies(df, columns=cat_columns)
```

Now that we don't have more missing values (you can double-check that) and categorical features are encoded, we can try to predict insurance claims. What accuracy can we get?

Building the model

We'll start by splitting the data into train and test datasets:

```
from sklearn.model_selection import train_test_split

labels = df.columns[2:]

X = df[labels]
y = df['target']

X_train, X_test, y_train, y_test = \
  train_test_split(X, y, test_size=0.05, random_state=RANDOM_SEED)
```

Our binary classification model is a Neural Network with batch normalization and dropout layers:

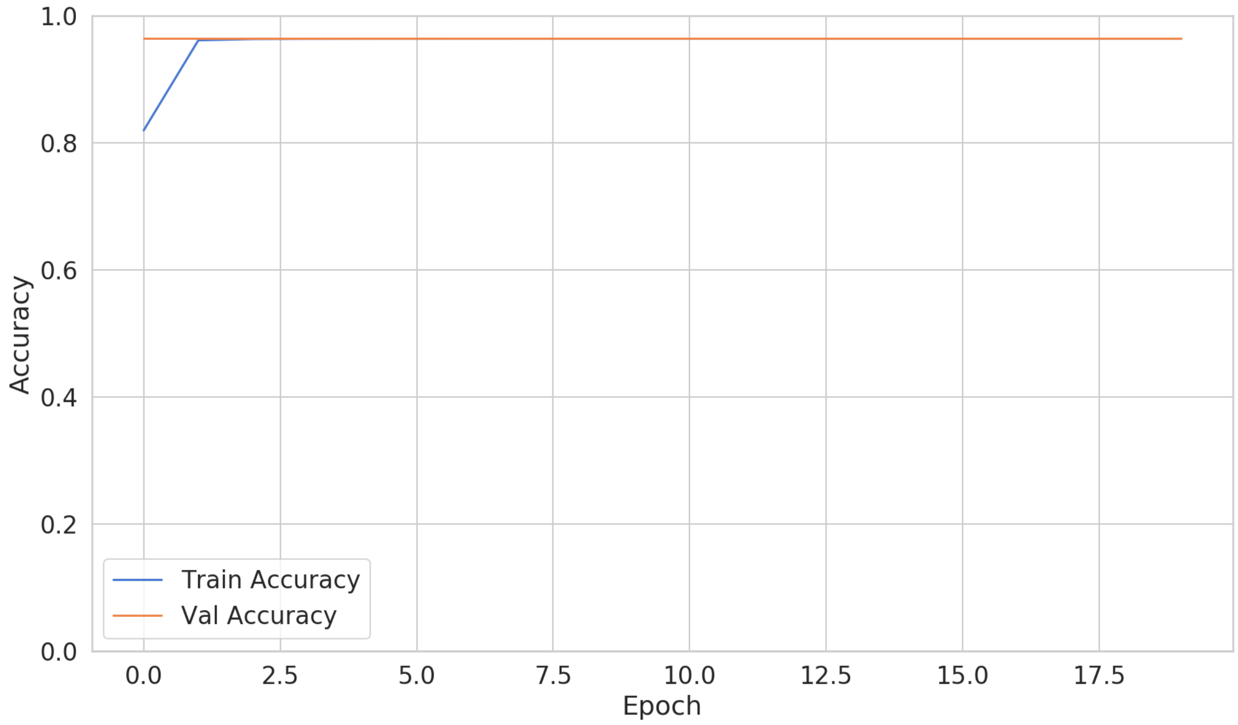
[136] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html

```
def build_model(train_data, metrics=["accuracy"]):
  model = keras.Sequential([
    keras.layers.Dense(
      units=36,
      activation='relu',
      input_shape=(train_data.shape[-1],)
    ),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.25),
    keras.layers.Dense(units=1, activation='sigmoid'),
  ])

  model.compile(
    optimizer=keras.optimizers.Adam(lr=0.001),
    loss=keras.losses.BinaryCrossentropy(),
    metrics=metrics
  )

  return model
```

You should be familiar with the training procedure:

```
BATCH_SIZE = 2048

model = build_model(X_train)
history = model.fit(
    X_train,
    y_train,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_split=0.05,
    shuffle=True,
    verbose=0
)
```

In general, you should strive for a small batch size (e.g. 32). Our case is a bit specific - we have highly imbalanced data, so we'll give a fair chance to each batch to contain some insurance claim data points.



The validation accuracy seems quite good. Let's evaluate the performance of our model:

```
model.evaluate(X_test, y_test, batch_size=BATCH_SIZE)
```

```
119043/119043 - loss: 0.1575 - accuracy: 0.9632
```

That's pretty good. It seems like our model is pretty awesome. Or is it?

```
def awesome_model_predict(features):
  return np.full((features.shape[0], ), 0)

y_pred = awesome_model_predict(X_test)
```

This **amazing** model predicts that there will be no claim, no matter the features. What accuracy does it get?

```
from sklearn.metrics import accuracy_score

accuracy_score(y_pred, y_test)
```

```
0.9632
```

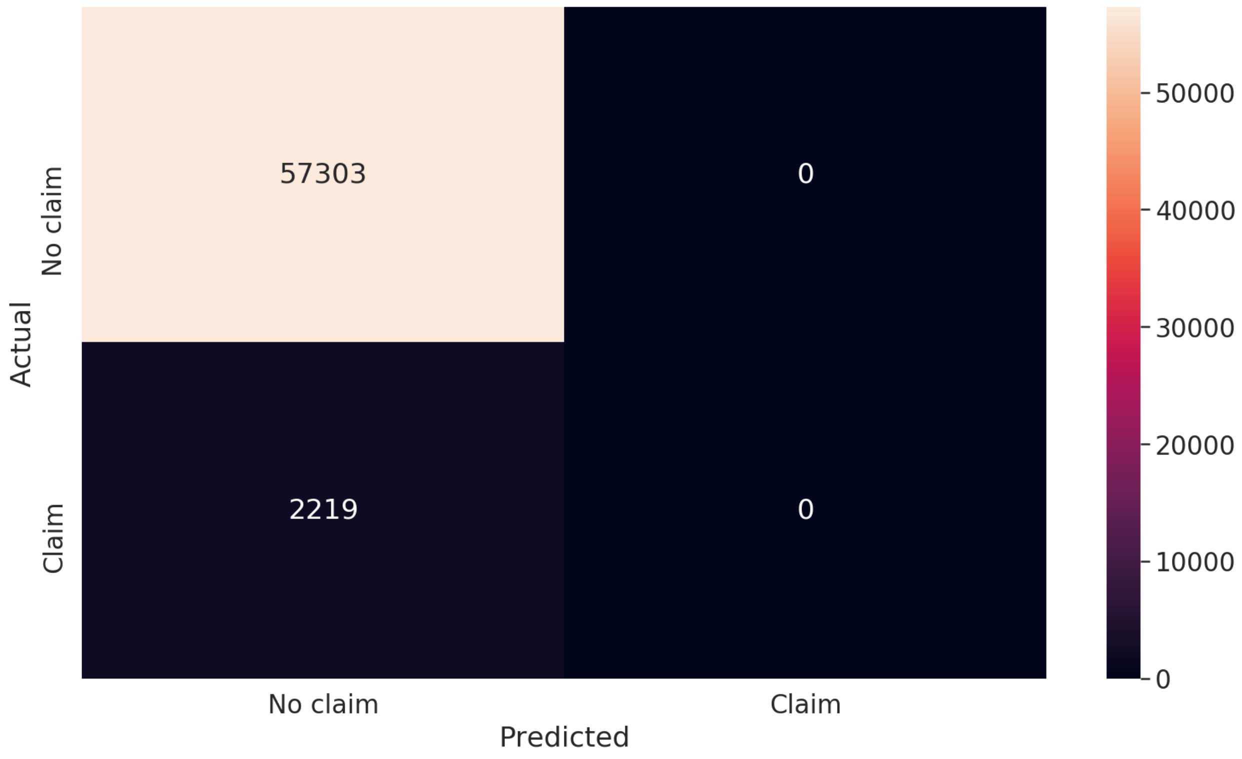
Sweet! Wait. What? This is as good as our complex model. Is there something wrong with our approach?

Evaluating the model

Not really. We're just using the wrong metric to evaluate our model. This is a well-known problem. The Accuracy paradox[137] suggests accuracy might not be the correct metric when the dataset is imbalanced. What can you do?

Using the correct metrics

One way to understand the performance of our model is to use a confusion matrix[138]. It shows us how well our model predicts for each class:



When the model is predicting everything perfectly, all values are on the main diagonal. That's not the case. So sad! Our complex model seems as dumb as dumb as our **awesome** model.

Good, now we know that our model is very bad at predicting insurance claims. Can we somehow tune it to do better?

[137] https://en.wikipedia.org/wiki/Accuracy_paradox
[138] https://en.wikipedia.org/wiki/Confusion_matrix

Useful metrics

We can use a wide range of other metrics to measure our peformance better:

- **Precision** - predicted positives divided by all positive predictions

$$\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

Low precision indicates a high number of false positives.

- **Recall** - percentage of actual positives that were correctly classified

$$\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

Low recall indicates a high number of false negatives.

- **F1 score** - combines precision and recall in one metric:

$$\frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

- **ROC curve** - A curve of True Positive Rate vs. False Positive Rate at different classification thresholds. It starts at (0,0) and ends at (1,1). A good model produces a curve that goes quickly from 0 to 1.
- **AUC (Area under the ROC curve)** - Summarizes the ROC curve with a single number. The best value is 1.0, while 0.5 is the worst.

Different combinations of precision and recall give you a better understanding of how well your model is performing for a given class:

- high precision + high recall : your model can be trusted when predicting this class
- high precision + low recall : you can trust the predictions for this class, but your model is not good at detecting it
- low precision + high recall: your model can detect the class but messes it up with other classes
- low precision + low recall : you can't trust the predictions for this class

Measuring your model

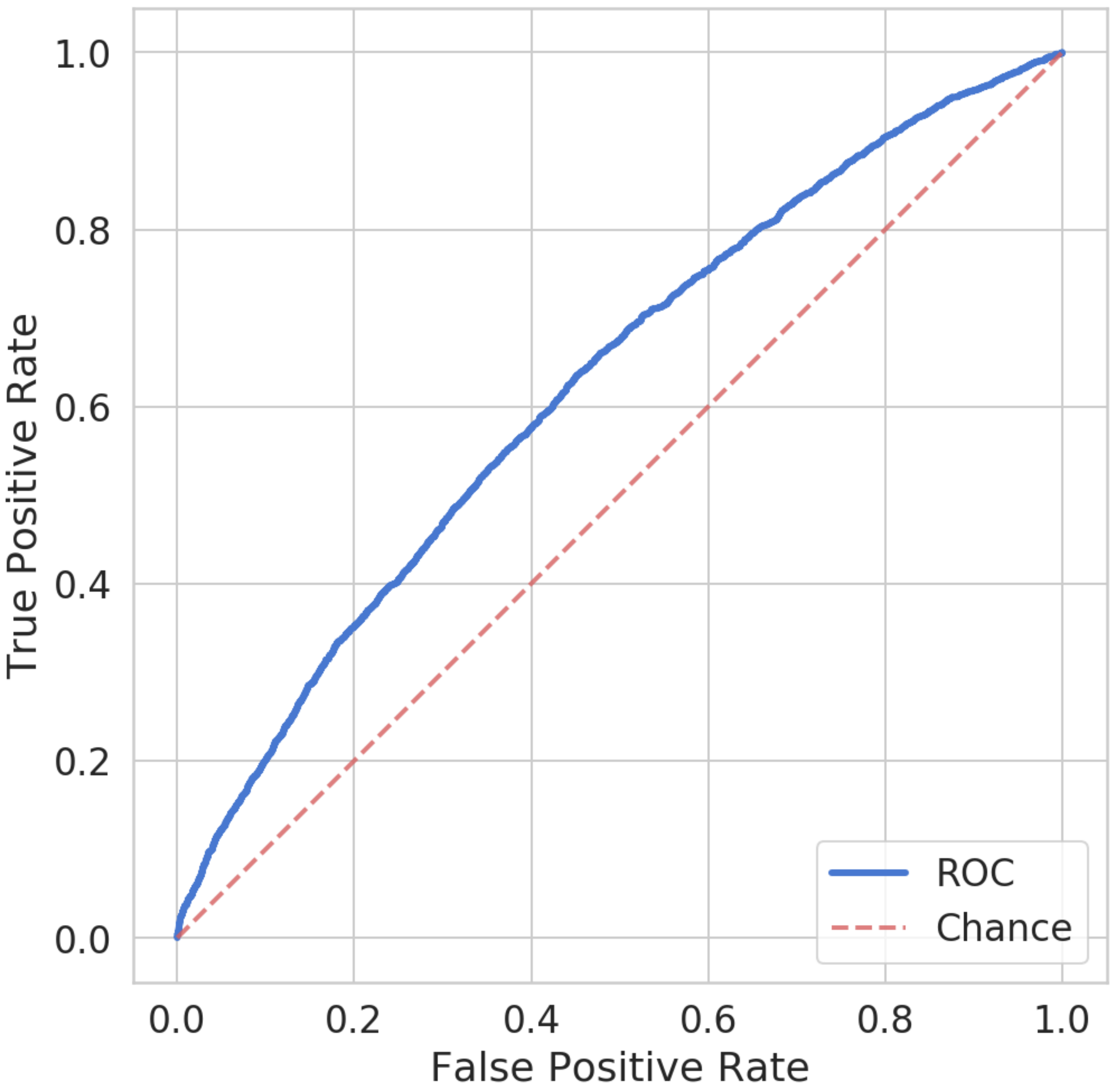
Luckily, Keras can calculate most of those metrics for you:

```
METRICS = [
      keras.metrics.TruePositives(name='tp'),
      keras.metrics.FalsePositives(name='fp'),
      keras.metrics.TrueNegatives(name='tn'),
      keras.metrics.FalseNegatives(name='fn'),
      keras.metrics.BinaryAccuracy(name='accuracy'),
      keras.metrics.Precision(name='precision'),
      keras.metrics.Recall(name='recall'),
      keras.metrics.AUC(name='auc'),
]
```

And here are the results:

```
loss :  0.1557253243213323
tp :  0.0
fp :  1.0
tn :  57302.0
fn :  2219.0
accuracy :  0.9627029
precision :  0.0
recall :  0.0
auc :  0.62021655
f1 score: 0.0
```

Here is the ROC:



Our model is complete garbage. And we can measure how much garbage it is. Can we do better?

Weighted model

We have many more examples of no insurance claims compared to those claimed. Let’s force our model to pay attention to the underrepresented class. We can do that by passing weights for each class. First we need to calcualte those:

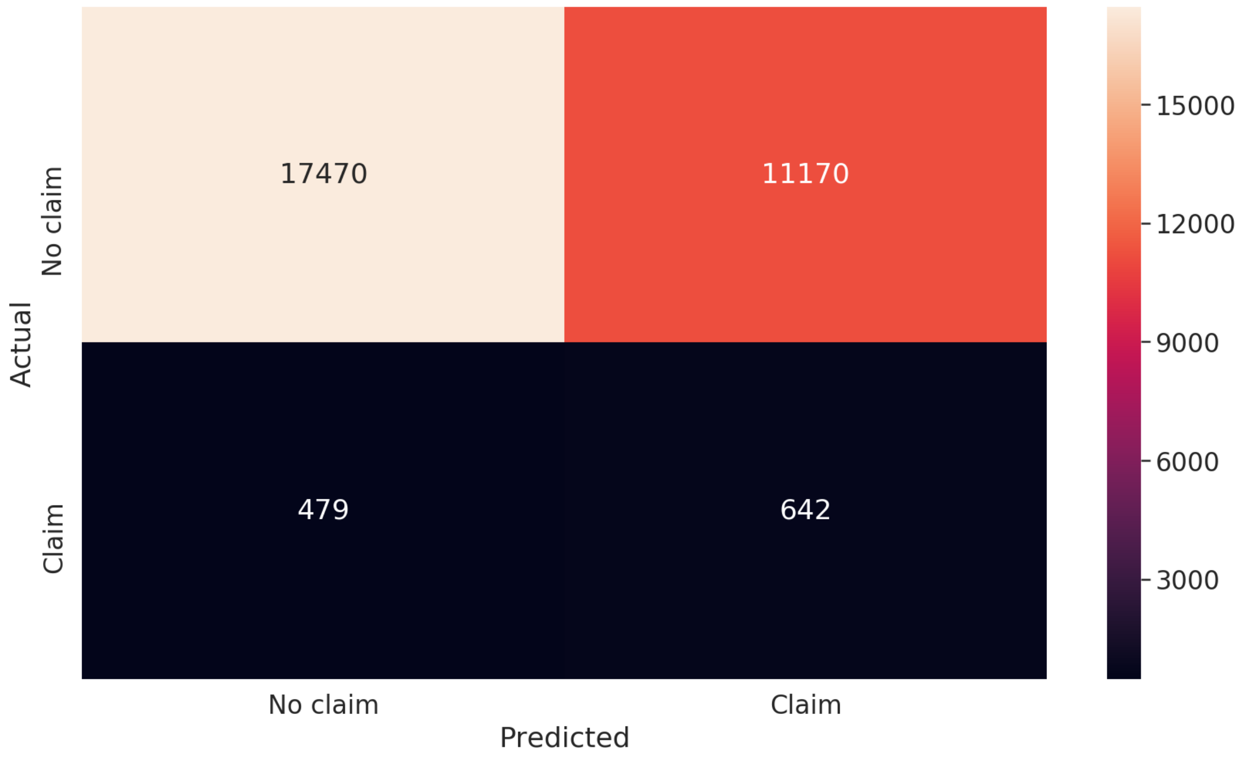
```
no_claim_count, claim_count = np.bincount(df.target)
total_count = len(df.target)

weight_no_claim = (1 / no_claim_count) * (total_count) / 2.0
weight_claim = (1 / claim_count) * (total_count) / 2.0

class_weights = {0: weight_no_claim, 1: weight_claim}
```

Now, let's use the weights when training our model:

```
model = build_model(X_train, metrics=METRICS)

history = model.fit(
    X_train,
    y_train,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_split=0.05,
    shuffle=True,
    verbose=0,
    class_weight=class_weights
)
```

Evaluation

Let's begin with the confusion matrix:



Things are a lot different now. We have a lot of correctly predicted insurance claims. The bad news is that we have a lot of predicted claims that were no claims. What can our metrics tell us?

```
loss :  0.6694403463347913
tp :  642.0
fp :  11170.0
tn :  17470.0
fn :  479.0
accuracy :  0.6085817
precision :  0.05435151
recall :  0.57270294
auc :  0.63104653
f1 score: 0.09928090930178612
```

The recall has jumped significantly while the precision bumped up only slightly. The F1-score is pretty low too! Overall, our model has improved somewhat. Especially, considering the minimal effort on our part. How can we do better?

Resampling techniques

These methods try to "correct" the balance in your data. They act as follows:

- oversampling - replicate examples from the under-represented class (claims)
- undersampling - sample from the most represented class (no claims) to keep only a few examples
- generate synthetic data - create new synthetic examples from the under-represented class

Naturally, a classifier trained on the "rebalanced" data will not know the original proportions. It is expected to have (much) lower accuracy since true proportions play a role in making a prediction.

You must think long and hard (that's what she said) before using resampling methods. It can be a perfectly good approach or complete nonsense.

Let's start by separating the classes:

```
X = pd.concat([X_train, y_train], axis=1)

no_claim = X[X.target == 0]
claim = X[X.target == 1]
```

Oversample minority class

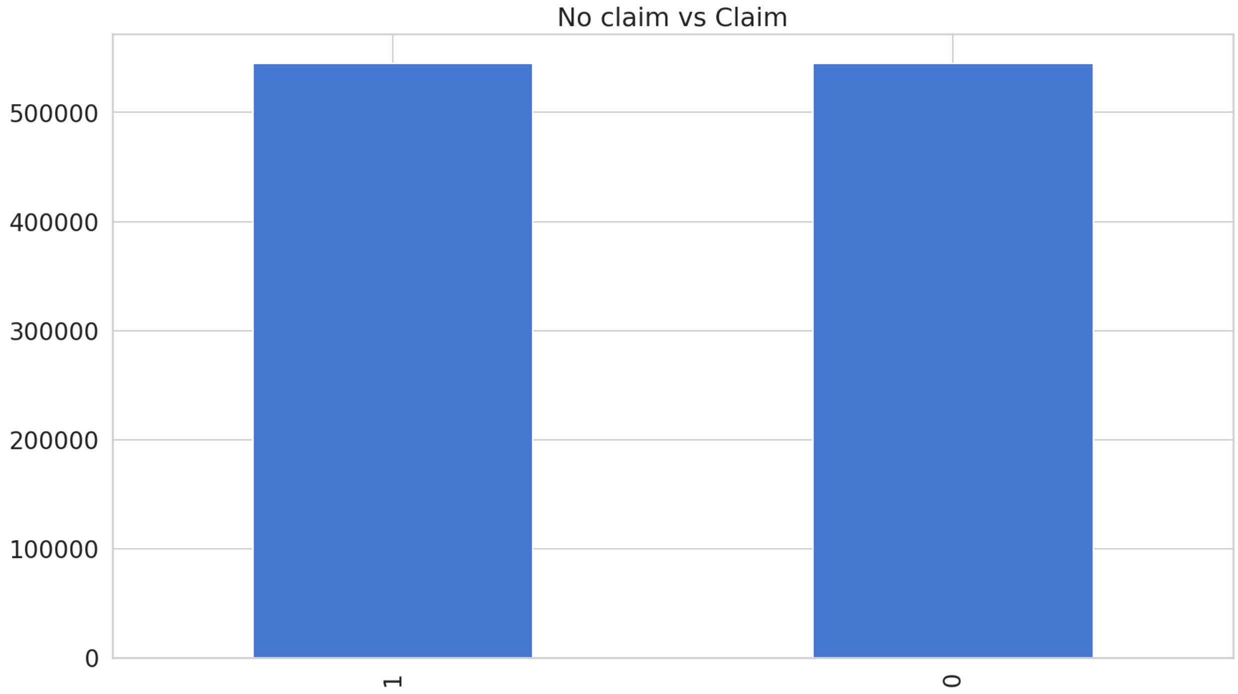
We'll start by adding more copies from the "insurance claim" class. This can be a good option when the data is limited. Either way, you might want to evaluate all approaches using your metrics.

We'll use the `resample()`[139] utility from scikit-learn:

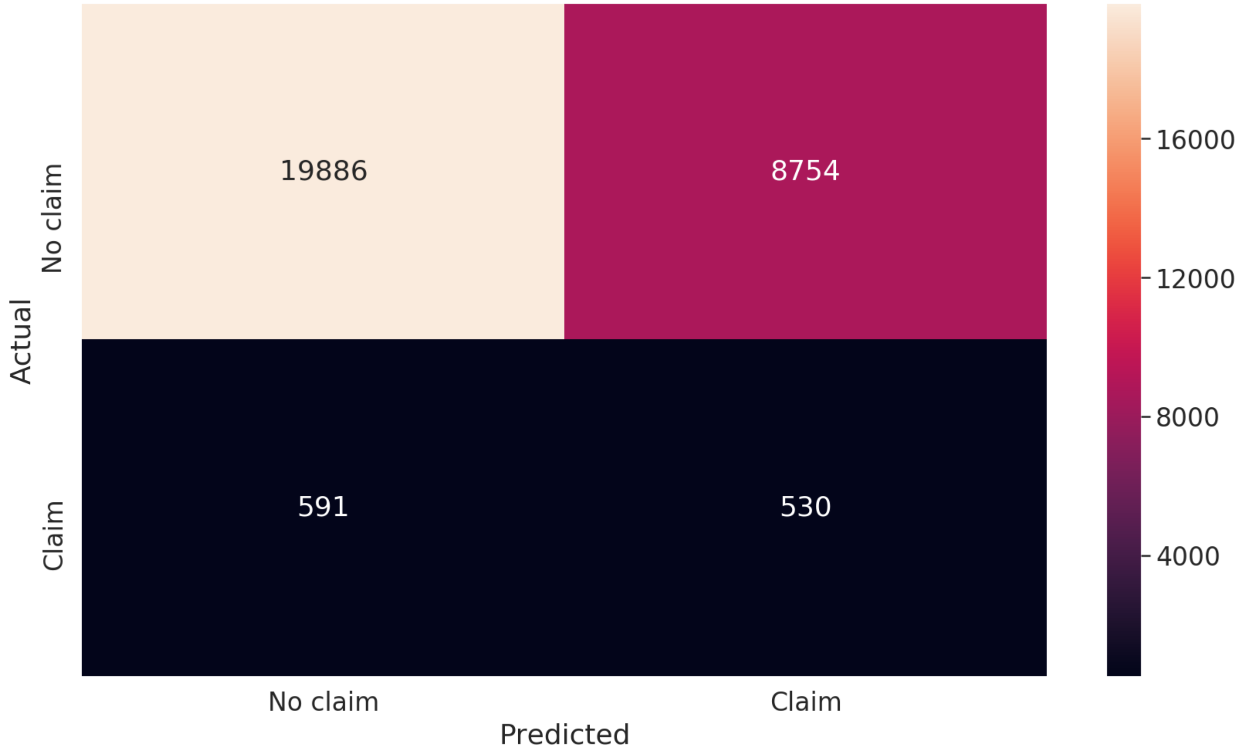
```
from sklearn.utils import resample

claim_upsampled = resample(claim,
                          replace=True,
                          n_samples=len(no_claim),
                          random_state=RANDOM_SEED)
```

Here is the new distribution of no claim vs claim:

[139] https://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html



Our new model performs like this:



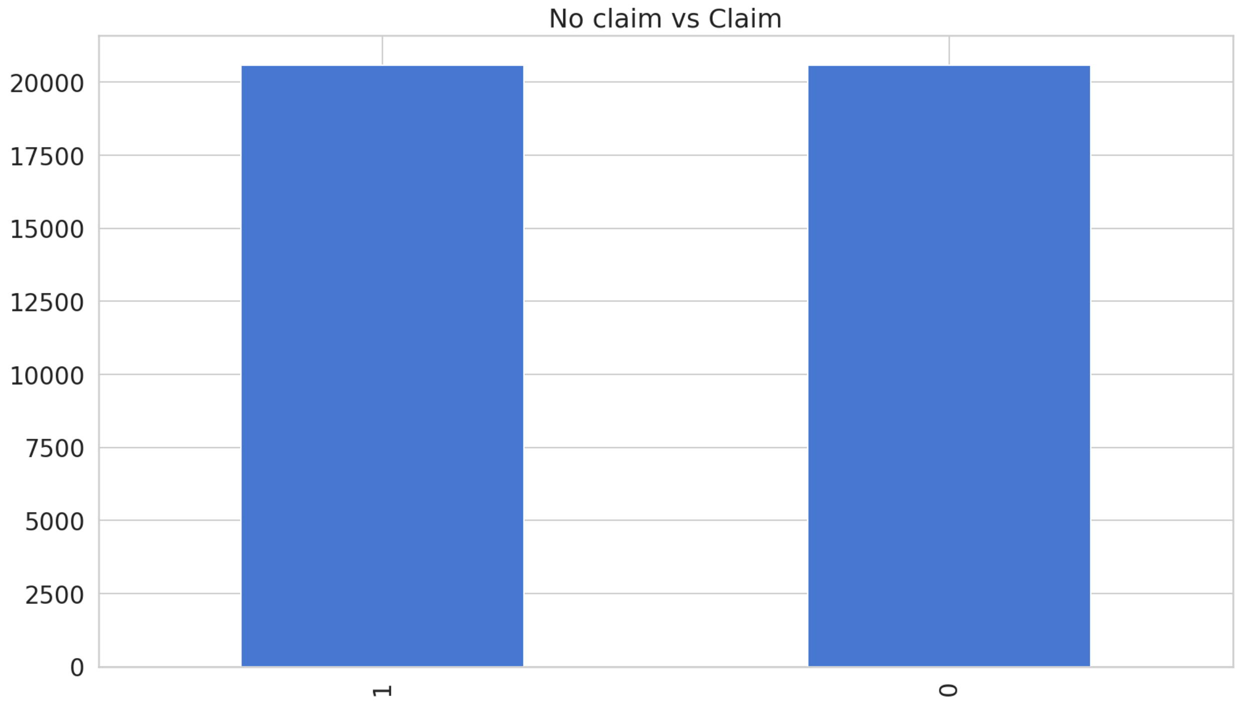
```
loss :  0.6123614118771424
tp :  530.0
fp :  8754.0
tn :  19886.0
fn :  591.0
accuracy :  0.68599844
precision :  0.057087462
recall :  0.47279215
auc :  0.6274258
f1 score: 0.10187409899086977
```

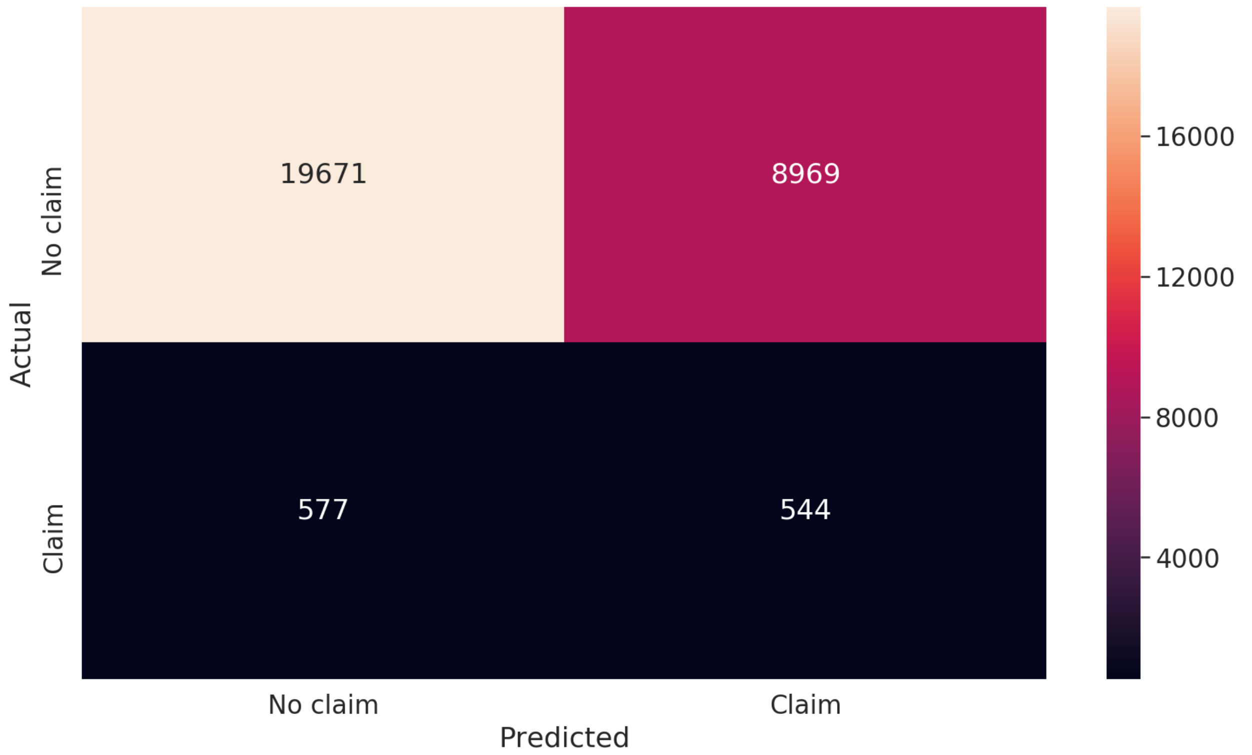
The performance of our model is similar to the weighted one. Can undersampling do better?

Undersample majority class

We'll remove samples from the no claim class and balance the data this way. This can be a good option when your dataset is large. Removing data can lead to underfitting on the test set.

```
no_claim_downsampled = resample(no_claim,
                                replace = False,
                                n_samples = len(claim),
                                random_state = RANDOM_SEED)
```





```
loss :  0.6377013992475753
tp :  544.0
fp :  8969.0
tn :  19671.0
fn :  577.0
accuracy :  0.67924464
precision :  0.057184905
recall :  0.485281
auc :  0.6206339
f1 score: 0.1023133345871732
```

Again, we don't have such impressive results but doing better than the baseline model.

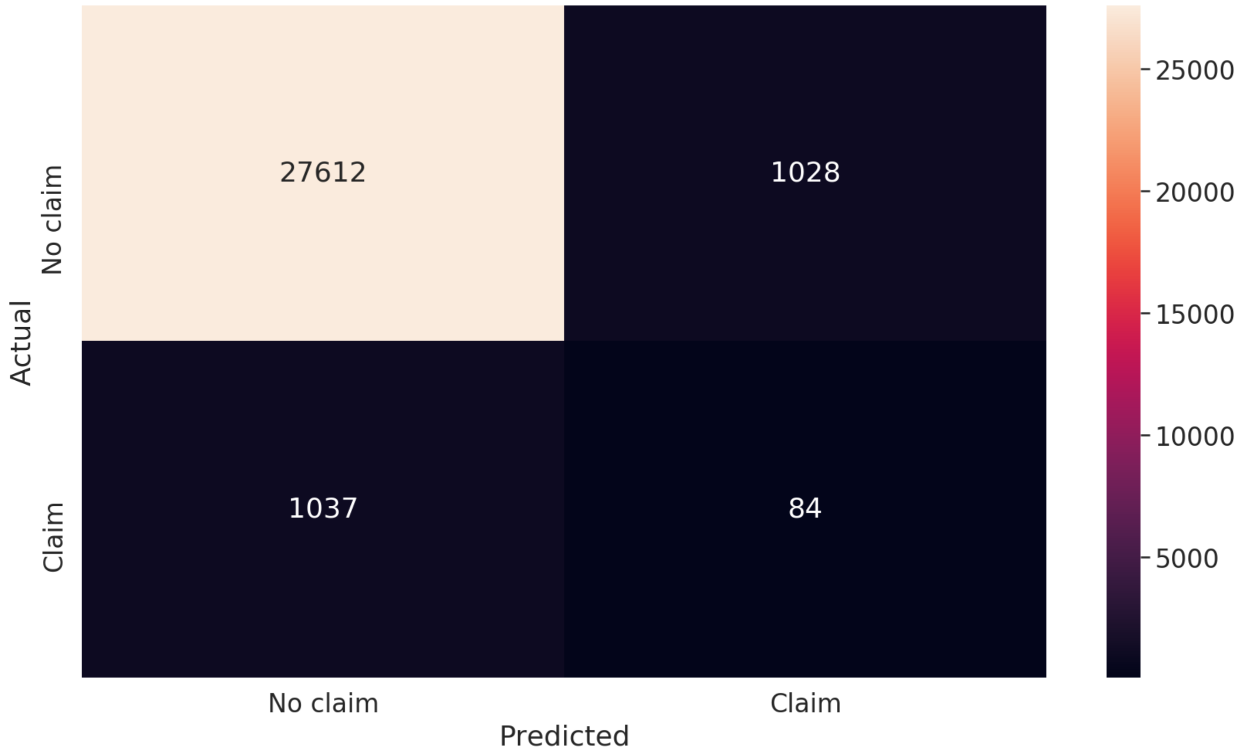
Generating synthetic samples

Let's try to simulate the data generation process by creating synthetic samples. We'll use the imbalanced-learn[140] library to do that.

One over-sampling method to generate synthetic data is the Synthetic Minority Oversampling Technique (SMOTE)[141]. It uses KNN algorithm to generate new data samples.

[140] http://imbalanced-learn.org
[141] https://arxiv.org/pdf/1106.1813.pdf

```python
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state=RANDOM_SEED, ratio=1.0)
X_train, y_train = sm.fit_sample(X_train, y_train)
```



```
loss :  0.26040001417683606
tp :  84.0
fp :  1028.0
tn :  27612.0
fn :  1037.0
accuracy :  0.9306139
precision :  0.07553957
recall :  0.0749331
auc :  0.5611229
f1 score: 0.07523510971786834
```

We have high accuracy but very low precision and recall. Not a useful approach for our dataset.

Conclusion

There are a lot of ways to handle imbalanced datasets. You should always start with something simple (like collecting more data or using a Tree-based model) and evaluate your model with the appropriate metrics. If all else fails, come back to this guide and try the more advanced approaches.

You learned how to:

- Impute missing data
- Handle categorical features
- Use the right metrics for classification tasks
- Set per class weights in Keras when training a model
- Use resampling techniques to balance the dataset

Run the complete code in your browser[142]

Remember that the best approach is almost always specific to the problem at hand (context is king). And sometimes, you can restate the problem as outlier/anomaly detection ;)

References

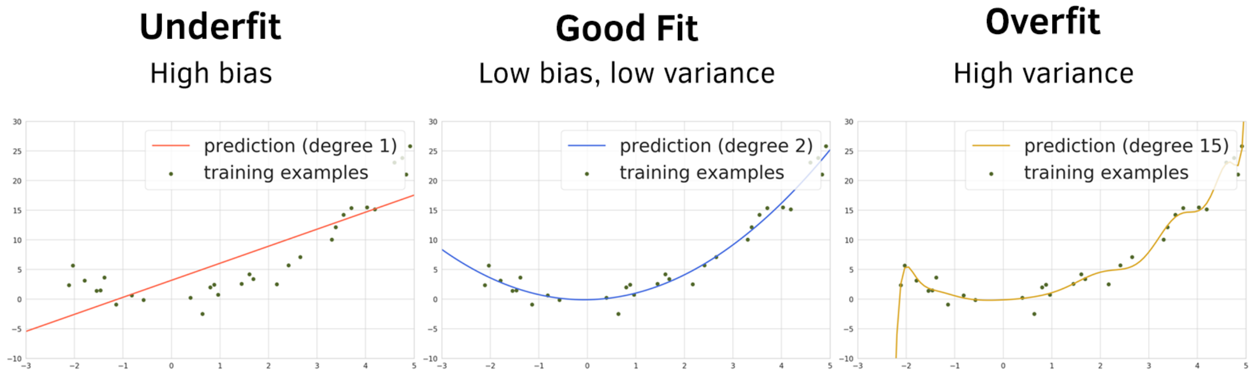
- Classification on imbalanced data[143]
- Dealing with Imbalanced Data[144]
- Resampling strategies for imbalanced datasets[145]
- imbalanced-learn - Tackle the Curse of Imbalanced Datasets in Machine Learning [146]

[142] https://colab.research.google.com/drive/1lZvXQxaO4mOT3-zImEkb0boDctju0cLw
[143] https://www.tensorflow.org/tutorials/structured_data/imbalanced_data
[144] https://towardsdatascience.com/methods-for-dealing-with-imbalanced-data-5b761be45a18
[145] https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets
[146] http://imbalanced-learn.org/en/stable/

Fixing Underfitting and Overfitting Models

> TL;DR Learn how to handle underfitting and overfitting models using TensorFlow 2, Keras and scikit-learn. Understand how you can use the bias-variance tradeoff to make better predictions.

The problem of the goodness of fit can be illustrated using the following diagrams:



Types of Model Fit

One way to describe the problem of underfitting is by using the concept of **bias**:

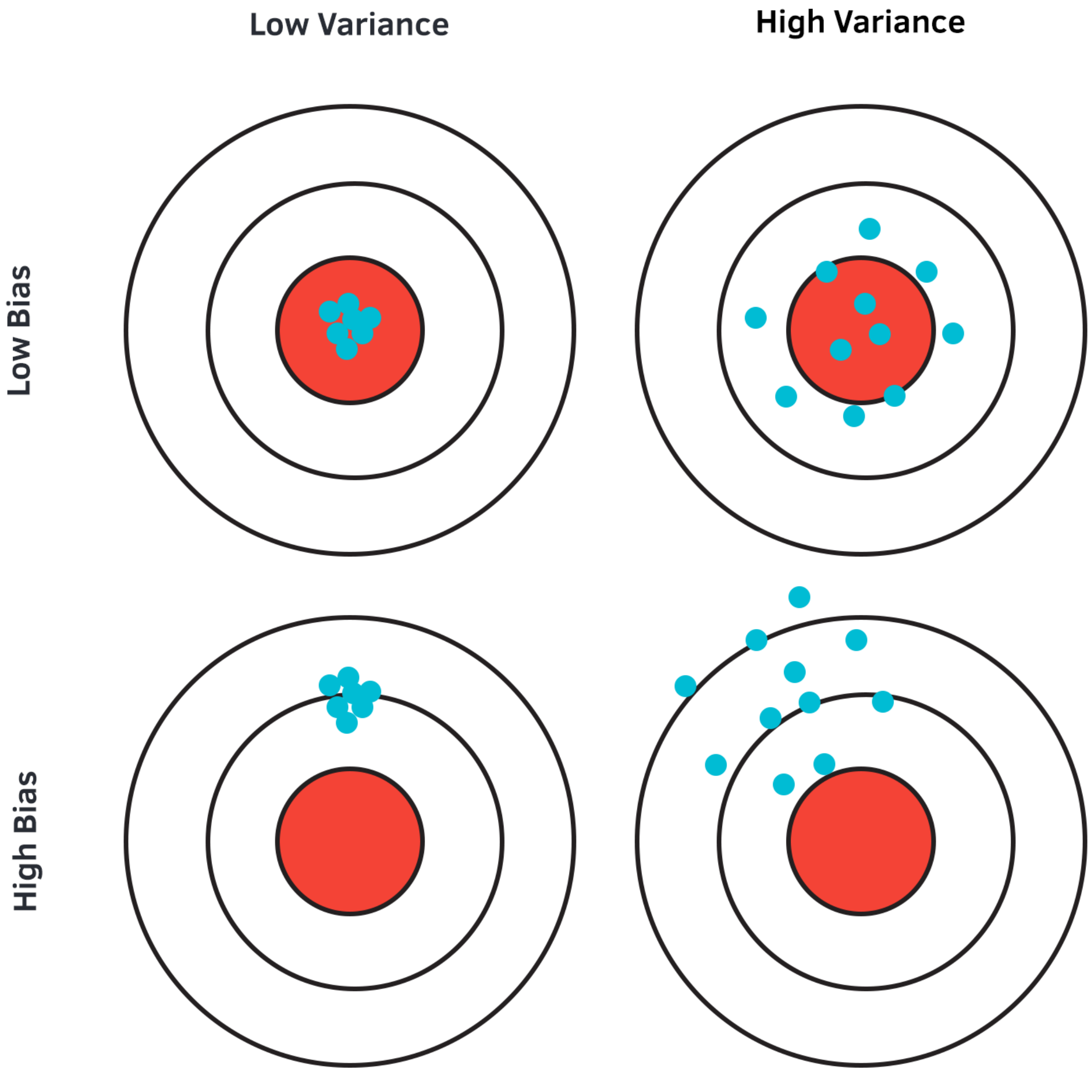
- a model has a **high bias** if it makes *a lot of mistakes on the training data*. We also say that the model **underfits**.
- a model has a **low bias** if *predicts well on the training data*

Naturally, we can use another concept to describe the problem of overfitting - **variance**:

- a model has a **high variance** if it *predicts very well on the training data but performs poorly on the test data*. Basically, overfitting means that the model has memorized the training data and can't generalize to things it hasn't seen.
- A model has a **low variance** if it generalizes well on the test data

Getting your model to *low bias and low variance can be pretty elusive* □. Nonetheless, we'll try to solve some of the common practical problems using a realistic dataset.

Here’s another way to look at the bias-variance tradeoff (heavily inspired by the original diagram of Andrew Ng):



Bias-Variance Tradeoff

You’ll learn how to diagnose and fix problems when:

- Your data has no predictive power
- Your model is too simple to make good predictions
- Your data brings the Curse of dimensionality
- Your model is too complex

Run the complete code in your browser[147]

Data

We'll use the Heart Disease dataset provided by UCI[148] and hosted on Kaggle[149]. Here is the description of the data:

> This database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to this date. The "goal" field refers to the presence of heart disease in the patient. It is integer valued from 0 (no presence) to 4.

We have 13 features and 303 rows of data. We're using those to predict whether or not a patient has heart disease.

Let's start with downloading and loading the data into a Pandas dataframe:

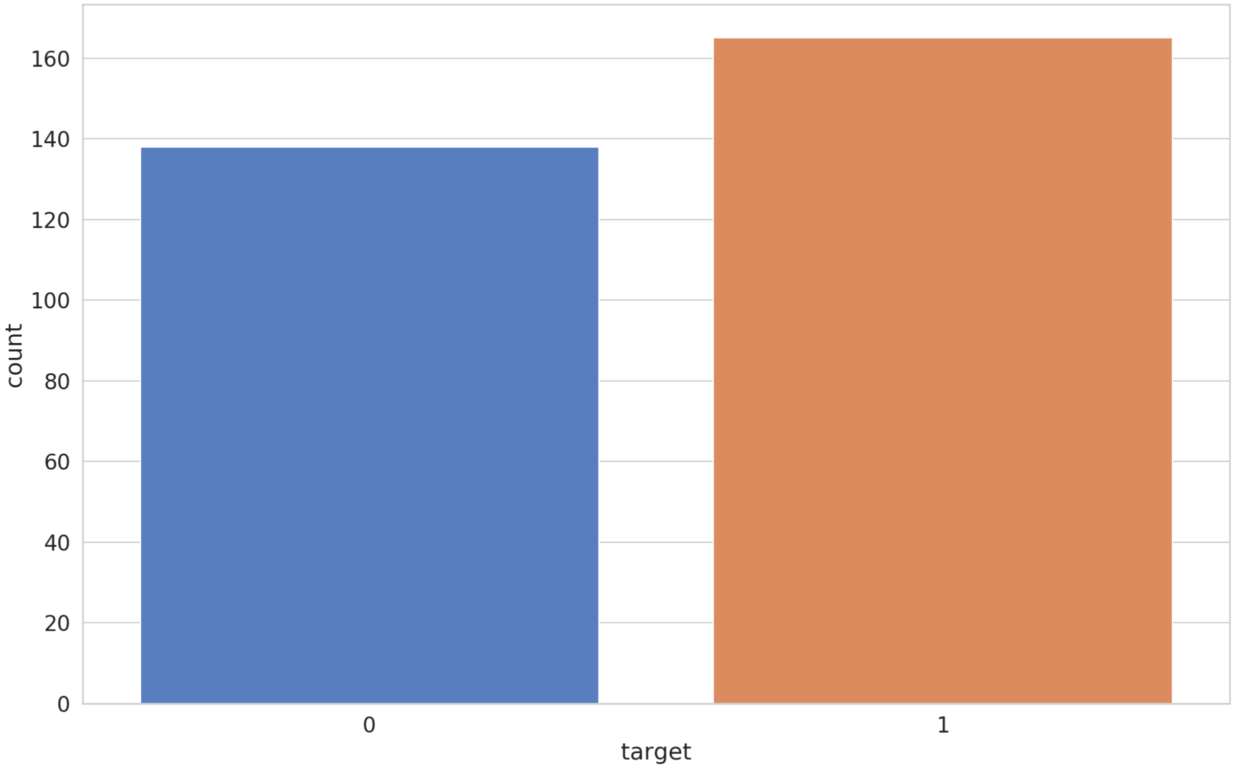
```
!pip install tensorflow-gpu
!pip install gdown

!gdown --id 1rsxu0CKFfI-xR1pH-5JQHcfZ7MIa08Q6 --output heart.csv
```

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

Exploration

We'll have a look at how well balanced the patients with and without heart disease are:

[147] https://colab.research.google.com/drive/19wKH_-4srUuJDRiZIqpE06tfXF3MLp0i
[148] https://archive.ics.uci.edu/ml/datasets/Heart+Disease
[149] https://www.kaggle.com/ronitf/heart-disease-uci

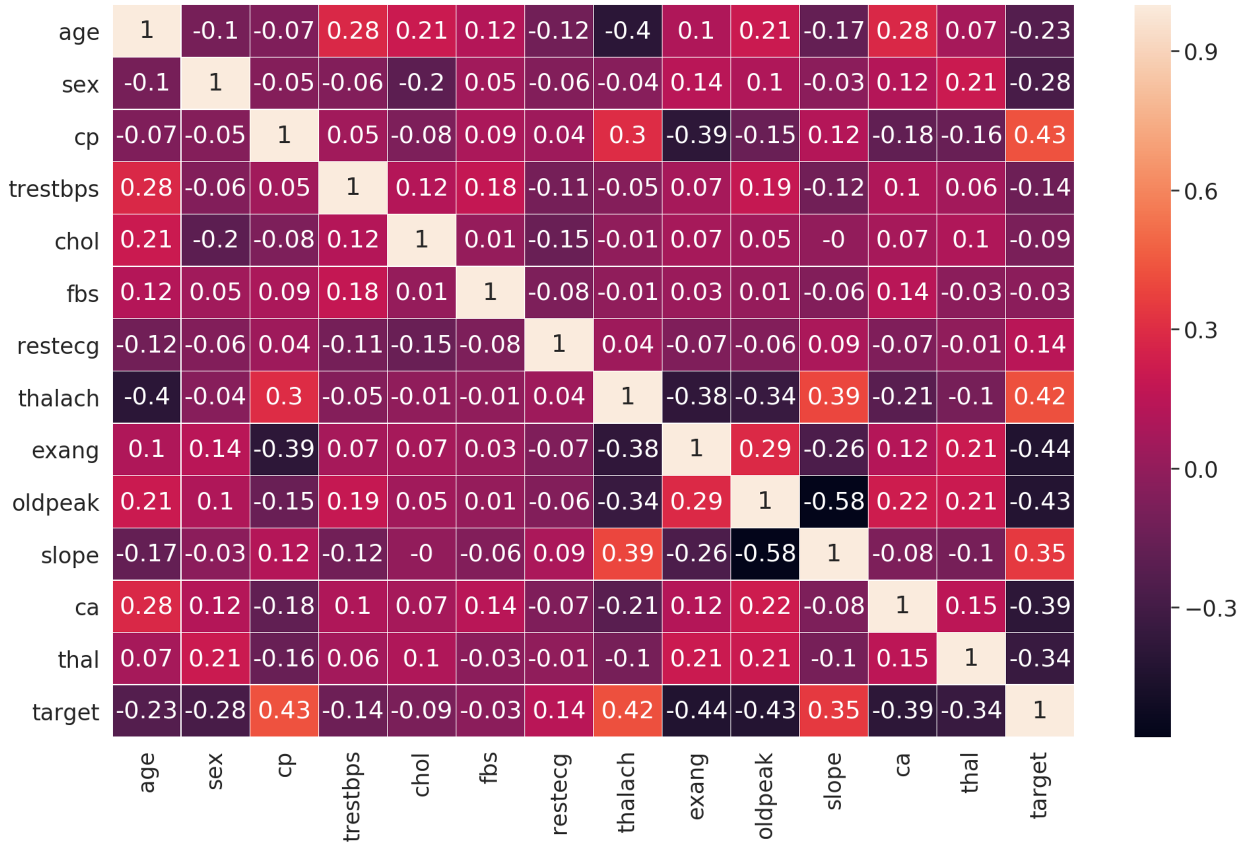


That looks pretty good. Almost no dataset will be perfectly balanced anyways. Do we have missing data?

```
df.isnull().values.any()
```

```
false
```

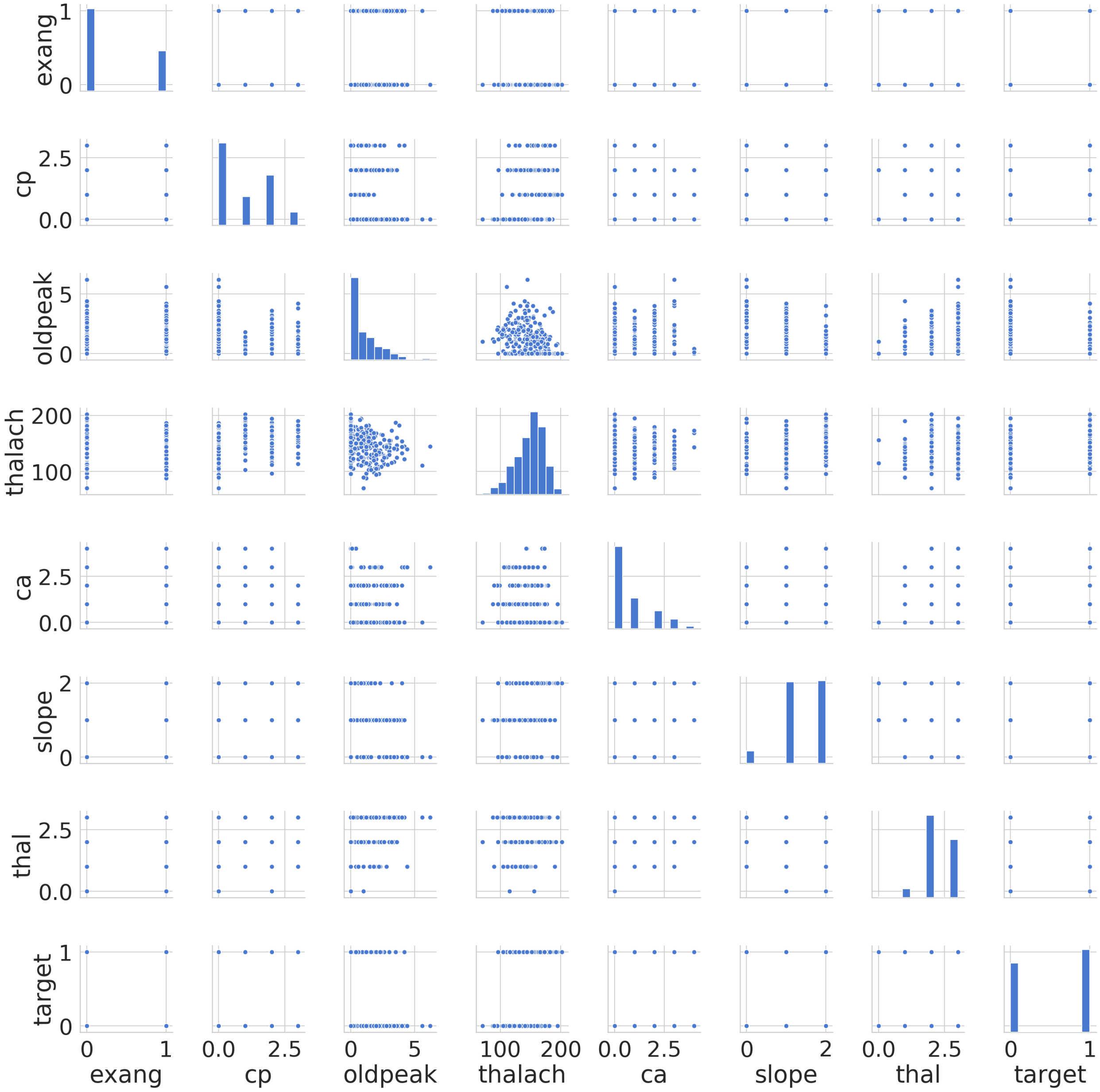
Nope. Let's have a look at the correlations between the features:

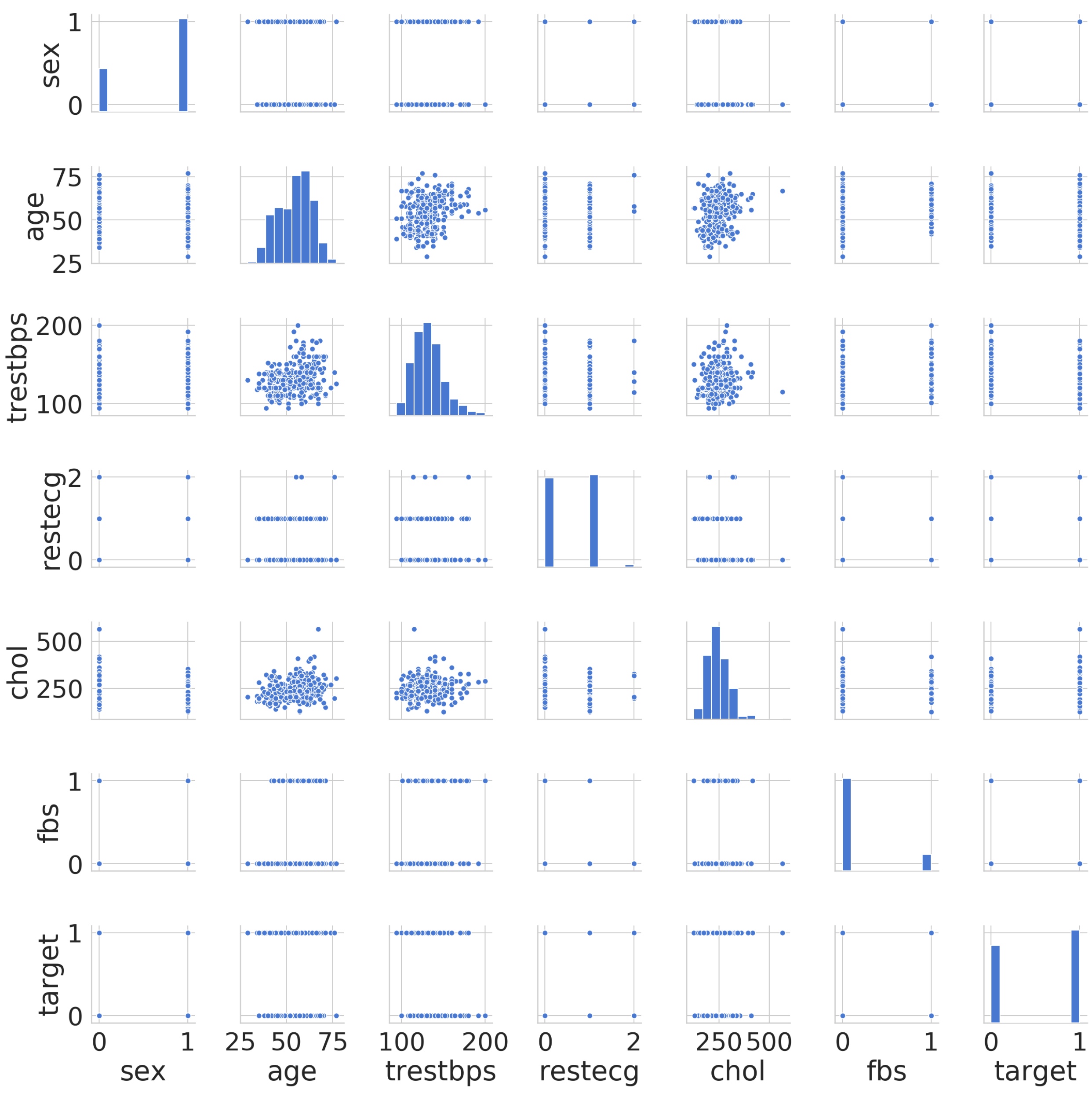


	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
age	1	-0.1	-0.07	0.28	0.21	0.12	-0.12	-0.4	0.1	0.21	-0.17	0.28	0.07	-0.23
sex	-0.1	1	-0.05	-0.06	-0.2	0.05	-0.06	-0.04	0.14	0.1	-0.03	0.12	0.21	-0.28
cp	-0.07	-0.05	1	0.05	-0.08	0.09	0.04	0.3	-0.39	-0.15	0.12	-0.18	-0.16	0.43
trestbps	0.28	-0.06	0.05	1	0.12	0.18	-0.11	-0.05	0.07	0.19	-0.12	0.1	0.06	-0.14
chol	0.21	-0.2	-0.08	0.12	1	0.01	-0.15	-0.01	0.07	0.05	-0	0.07	0.1	-0.09
fbs	0.12	0.05	0.09	0.18	0.01	1	-0.08	-0.01	0.03	0.01	-0.06	0.14	-0.03	-0.03
restecg	-0.12	-0.06	0.04	-0.11	-0.15	-0.08	1	0.04	-0.07	-0.06	0.09	-0.07	-0.01	0.14
thalach	-0.4	-0.04	0.3	-0.05	-0.01	-0.01	0.04	1	-0.38	-0.34	0.39	-0.21	-0.1	0.42
exang	0.1	0.14	-0.39	0.07	0.07	0.03	-0.07	-0.38	1	0.29	-0.26	0.12	0.21	-0.44
oldpeak	0.21	0.1	-0.15	0.19	0.05	0.01	-0.06	-0.34	0.29	1	-0.58	0.22	0.21	-0.43
slope	-0.17	-0.03	0.12	-0.12	-0	-0.06	0.09	0.39	-0.26	-0.58	1	-0.08	-0.1	0.35
ca	0.28	0.12	-0.18	0.1	0.07	0.14	-0.07	-0.21	0.12	0.22	-0.08	1	0.15	-0.39
thal	0.07	0.21	-0.16	0.06	0.1	-0.03	-0.01	-0.1	0.21	0.21	-0.1	0.15	1	-0.34
target	-0.23	-0.28	0.43	-0.14	-0.09	-0.03	0.14	0.42	-0.44	-0.43	0.35	-0.39	-0.34	1

Features like cp (chest pain type), exang (exercise induced angina), and oldpeak (ST depression induced by exercise relative to rest) seem to have a decent correlation with our target variable.

Let’s have a look at the distributions of our features, starting with the most correlated to the target variable:


exang
cp
oldpeak
thalach
ca
slope
thal
target
0
1
0.0
2.5
5
100
200
2



Seems like only oldpeak is a non-categorical feature. It appears that the data contains several features with outliers. You might want to explore those on your own, if interested :)

Underfitting

We'll start by building a couple of models that underfit and proceed by fixing the issue in some way.

Recall that your model underfits when it makes mistakes on the training data. Here are the most common reasons for that:

- The data features are not informative
- Your model is too simple to predict the data (e.g. linear model predicts non-linear data)

Data with no predictive power

We'll build a model with the *trestbps (resting blood pressure)* feature. Its correlation with the target variable is low: -0.14. Let's prepare the data:

```
from sklearn.model_selection import train_test_split

X = df[['trestbps']]
y = df.target

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

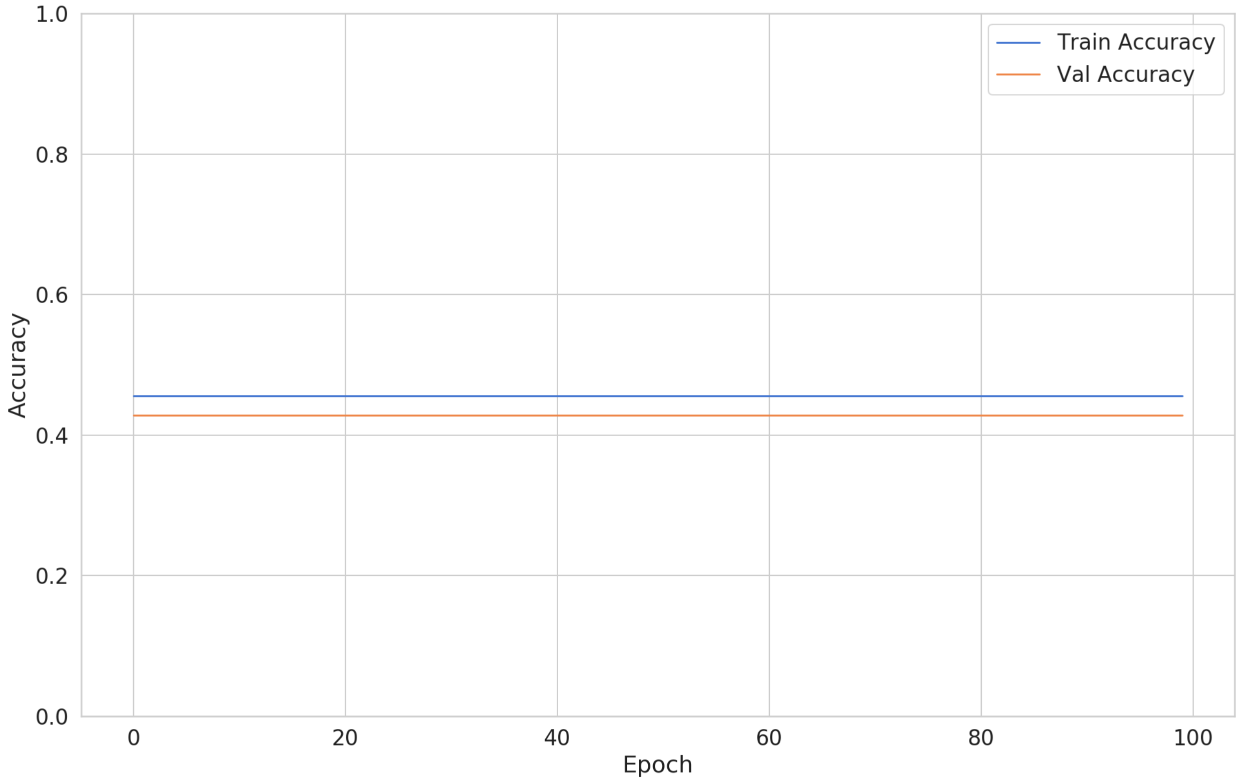
We'll build a binary classifier with 2 hidden layers:

```
def build_classifier(train_data):
  model = keras.Sequential([
    keras.layers.Dense(
      units=32,
      activation='relu',
      input_shape=[train_data.shape[1]]
    ),
    keras.layers.Dense(units=16, activation='relu'),
    keras.layers.Dense(units=1),
  ])

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

  return model
```

And train it for 100 epochs:

```
BATCH_SIZE = 32

clf = build_classifier(X_train)

clf_history = clf.fit(
  x=X_train,
  y=y_train,
  shuffle=True,
  epochs=100,
  validation_split=0.2,
  batch_size=BATCH_SIZE,
  verbose=0
)
```

Here’s how the train and validation accuracy changes during training:



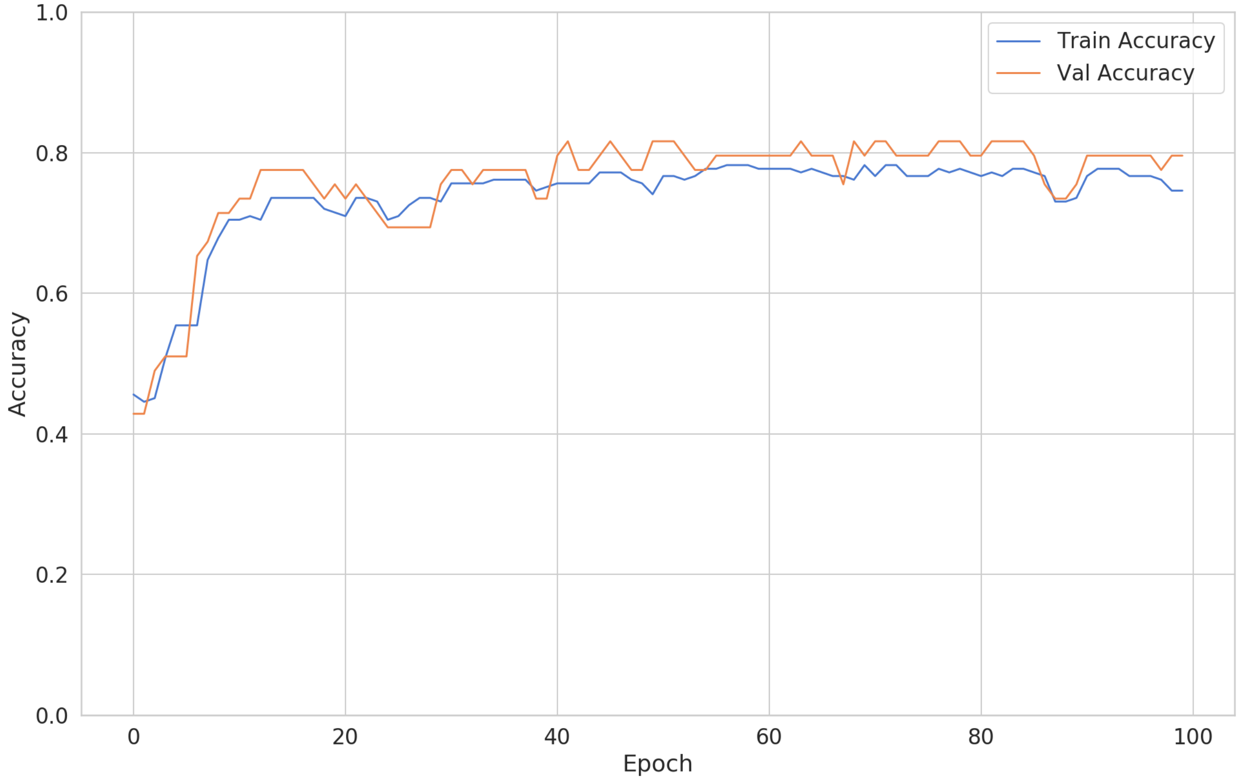
Our model is flatlining. This is expected, the feature we’re using has no predictive power.

The fix

Knowing that we’re using an uninformative feature makes it easy to fix the issue. We can use other feature(s):

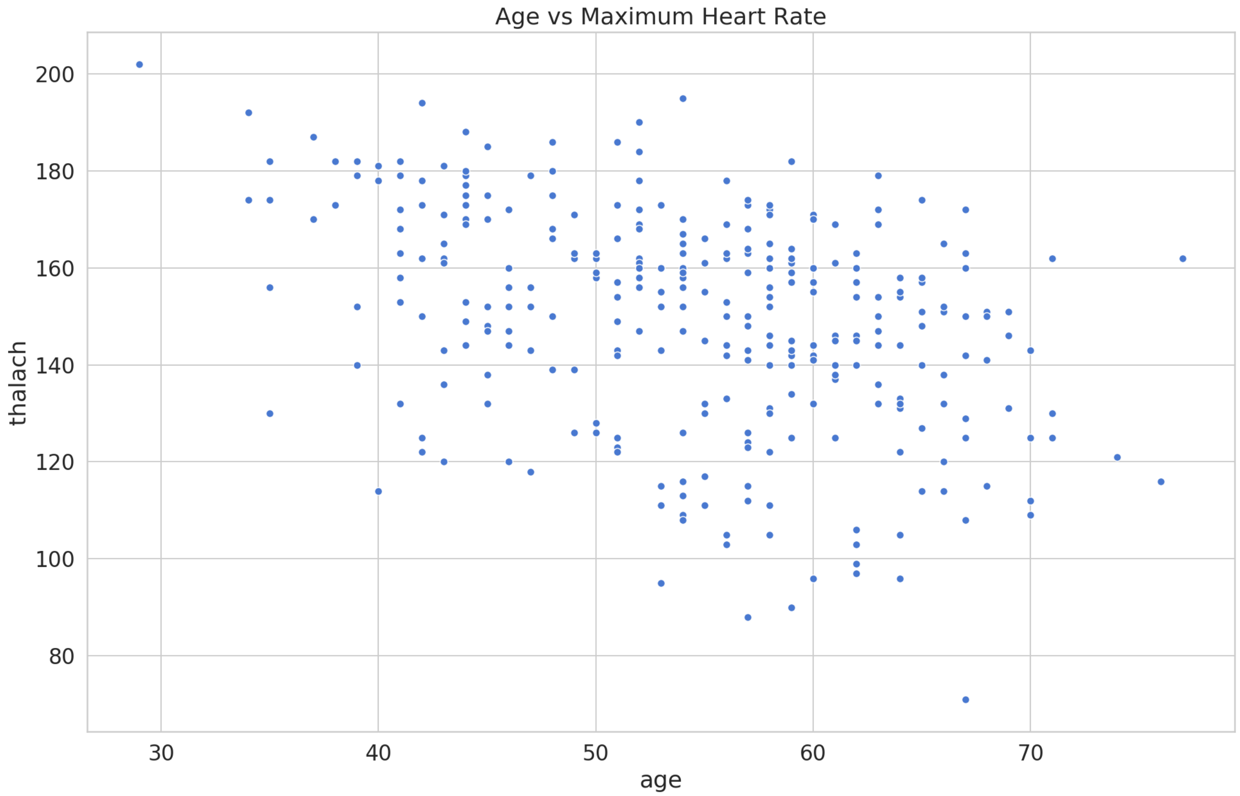
```
X = pd.get_dummies(df[['oldpeak', 'cp']], columns=["cp"])
y = df.target

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

And here are the results (using the same model, created from scratch):



Underpowered model

In this case, we're going to build a regressive model and try to predict the patient maximum heart rate (thalach) from its age.



Before starting our analysis, we'll use MinMaxScaler[150] from scikit-learn to scale the feature values in the 0-1 range:

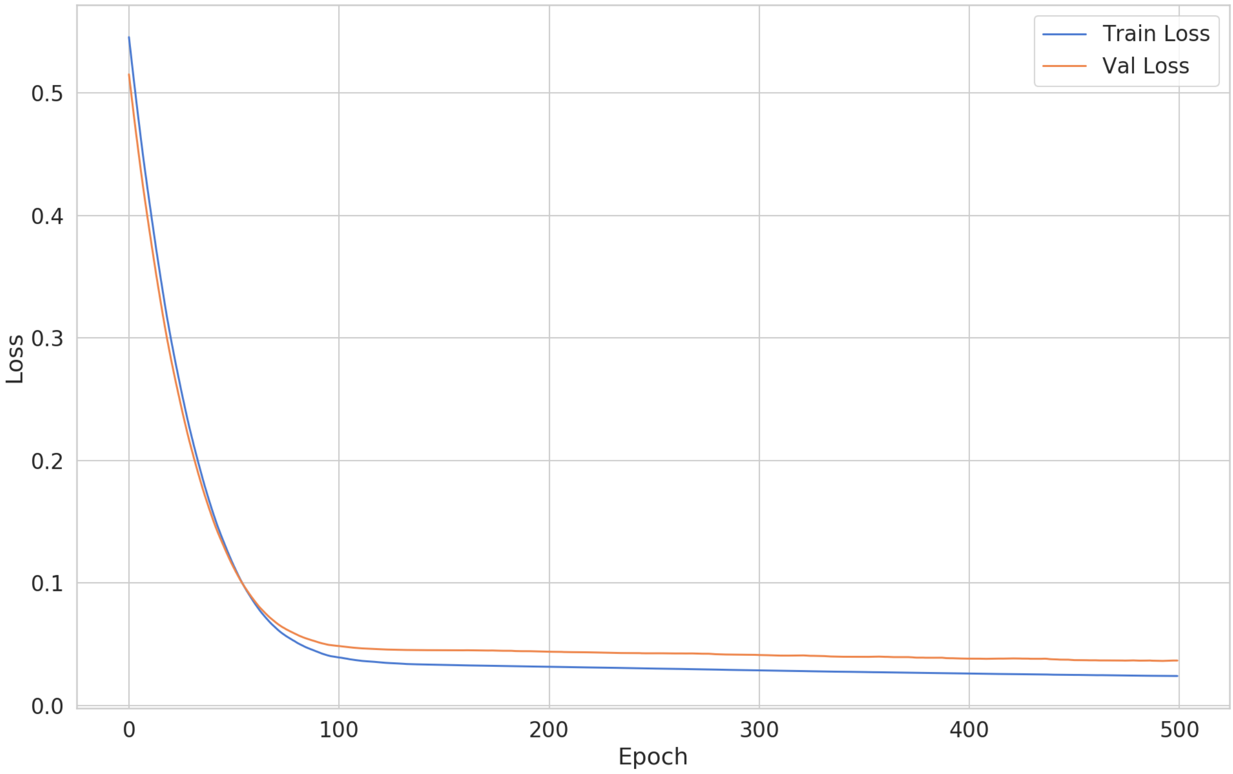
```
from sklearn.preprocessing import MinMaxScaler

s = MinMaxScaler()

X = s.fit_transform(df[['age']])
y = s.fit_transform(df[['thalach']])

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

Our model is a simple linear regression:

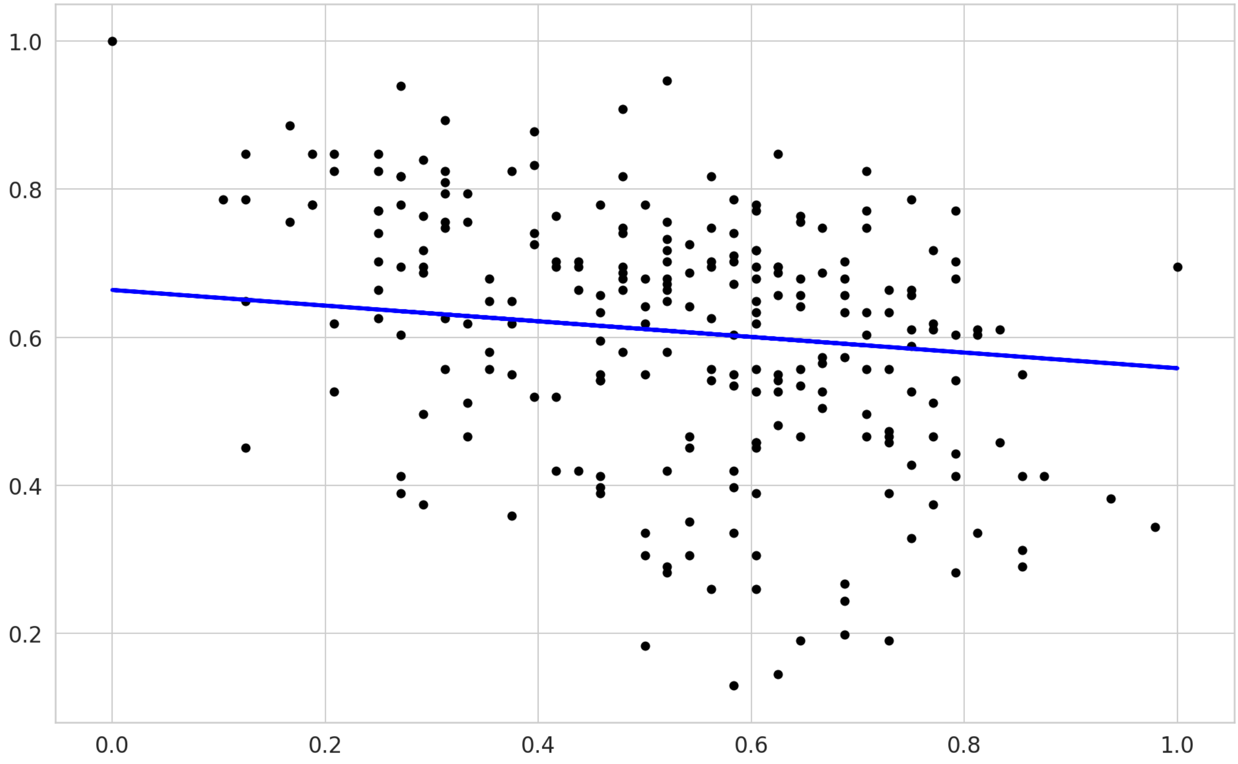
[150] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

```
lin_reg = keras.Sequential([
  keras.layers.Dense(
    units=1,
    activation='linear',
    input_shape=[X_train.shape[1]]
  ),
])

lin_reg.compile(
  loss="mse",
  optimizer="adam",
  metrics=['mse']
)
```

Here's the train/validation loss:



Here are the predictions from our model:



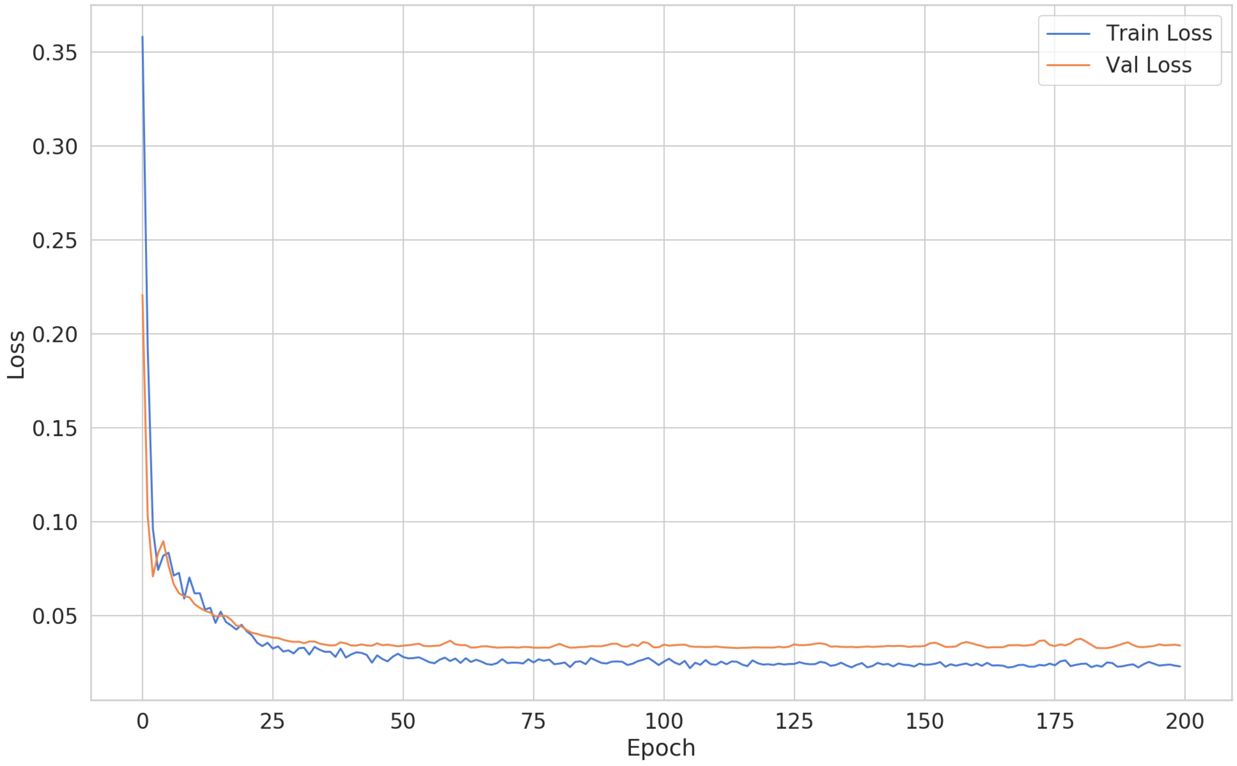
You can kinda see that a linear model might not be the perfect fit here.

The fix

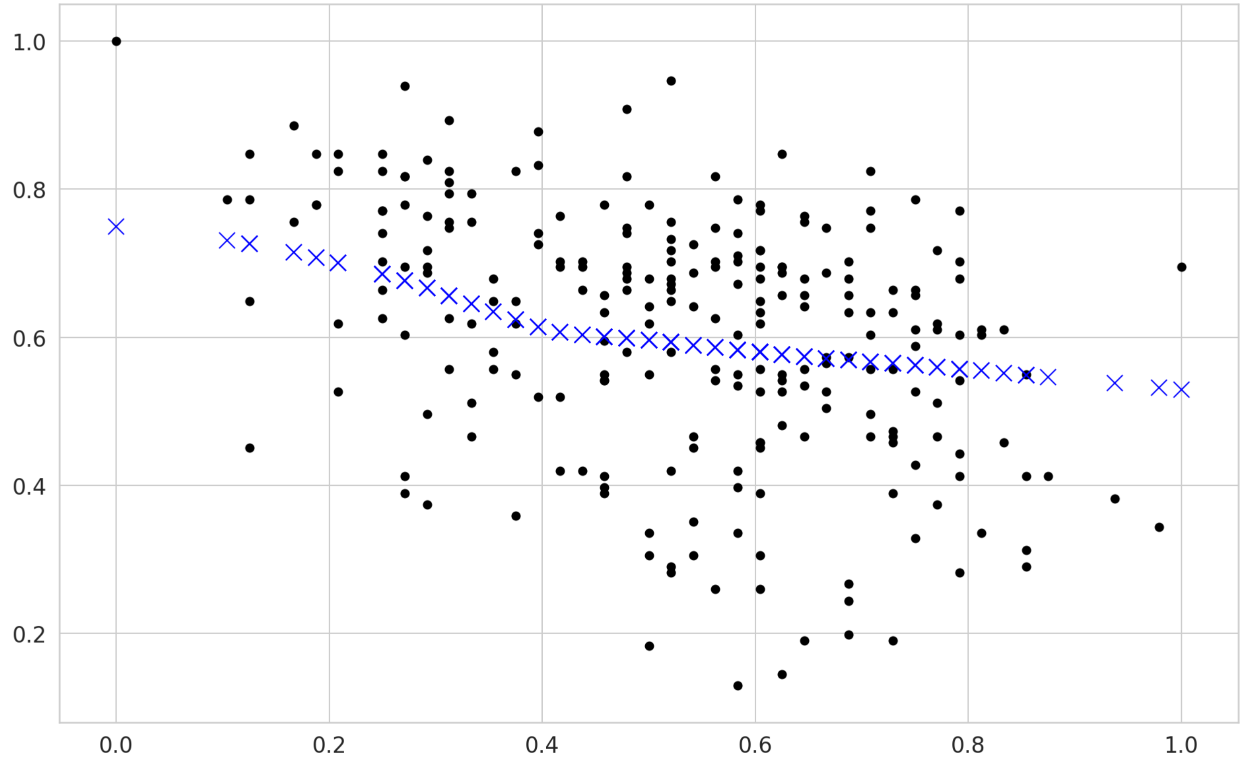
We'll use the same training process, except that our model is going to be a lot more complex:

```
lin_reg = keras.Sequential([
  keras.layers.Dense(
    units=64,
    activation='relu',
    input_shape=[X_train.shape[1]]
  ),
  keras.layers.Dropout(rate=0.2),
  keras.layers.Dense(units=32, activation='relu'),
  keras.layers.Dropout(rate=0.2),
  keras.layers.Dense(units=16, activation='relu'),
  keras.layers.Dense(units=1, activation='linear'),
])

lin_reg.compile(
  loss="mse",
  optimizer="adam",
  metrics=['mse']
)
```

Here's the training/validation loss:



Our validation loss is similar. What about the predictions:



Interesting, right? Our model broke from the linear-only predictions. Note that this fix included adding more parameters and increasing the regularization (using Dropout).

Overfitting

A model overfits when predicts training data well but performs poor on the validation set. Here are some of the reasons for that:

- Your data has many features but a small number of examples (curse of dimensionality)
- Your model is too complex for the data (Early stopping)

Curse of dimensionality

The Curse of dimensionality[151] refers to the problem of having too many features (dimensions), compared to the data points (examples). The most common way to solve this problem is to add more information.

We'll use a couple of features to create our dataset:

[151] https://en.wikipedia.org/wiki/Curse_of_dimensionality

```
X = df[['oldpeak', 'age', 'exang', 'ca', 'thalach']]
X = pd.get_dummies(X, columns=['exang', 'ca', 'thalach'])
y = df.target

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

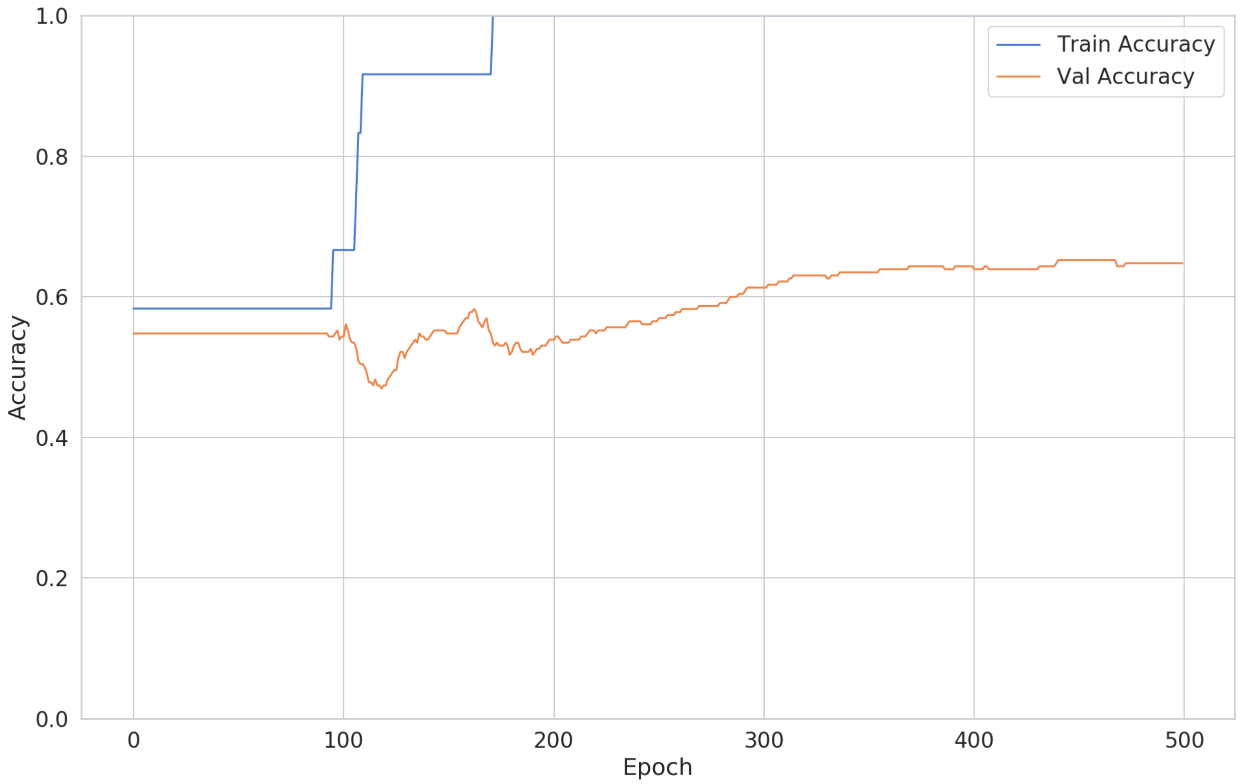
Our model contains one hidden layer:

```
def build_classifier():

  model = keras.Sequential([
    keras.layers.Dense(
      units=16,
      activation='relu',
      input_shape=[X_train.shape[1]]
    ),
    keras.layers.Dense(units=1, activation='sigmoid'),
  ])

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

  return model
```

Here’s the interesting part. We’re using just a tiny bit of the data for training:

```
clf = build_classifier()

clf_history = clf.fit(
  x=X_train,
  y=y_train,
  shuffle=True,
  epochs=500,
  validation_split=0.95,
  batch_size=BATCH_SIZE,
  verbose=0
)
```

Here’s the result of the training:



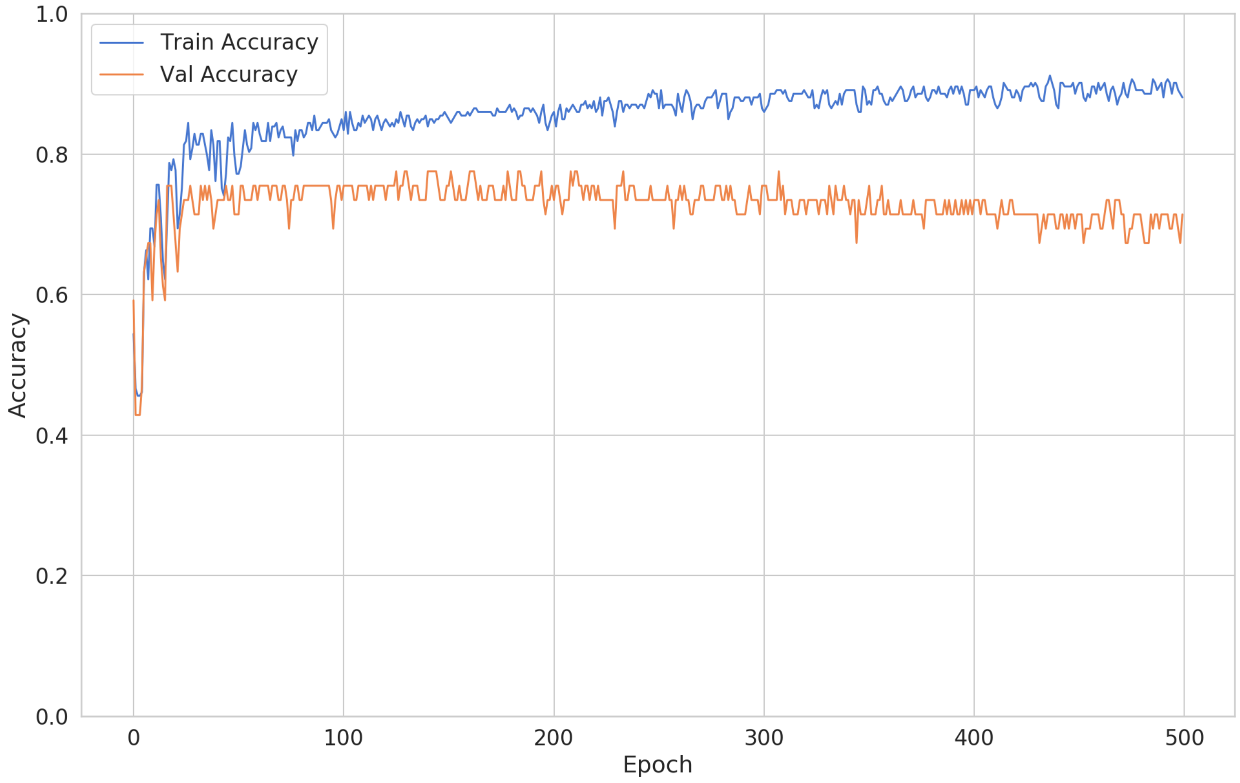
The fix

Our solution will be pretty simple - add more data. However, you can provide additional information via other methods (i.e. Bayesian prior) or reduce the number of features via feature selection.

Let’s try the simple approach:

```
clf = build_classifier()

clf_history = clf.fit(
  x=X_train,
  y=y_train,
  shuffle=True,
  epochs=500,
  validation_split=0.2,
  batch_size=BATCH_SIZE,
  verbose=0
)
```

The training/validation loss looks like this:



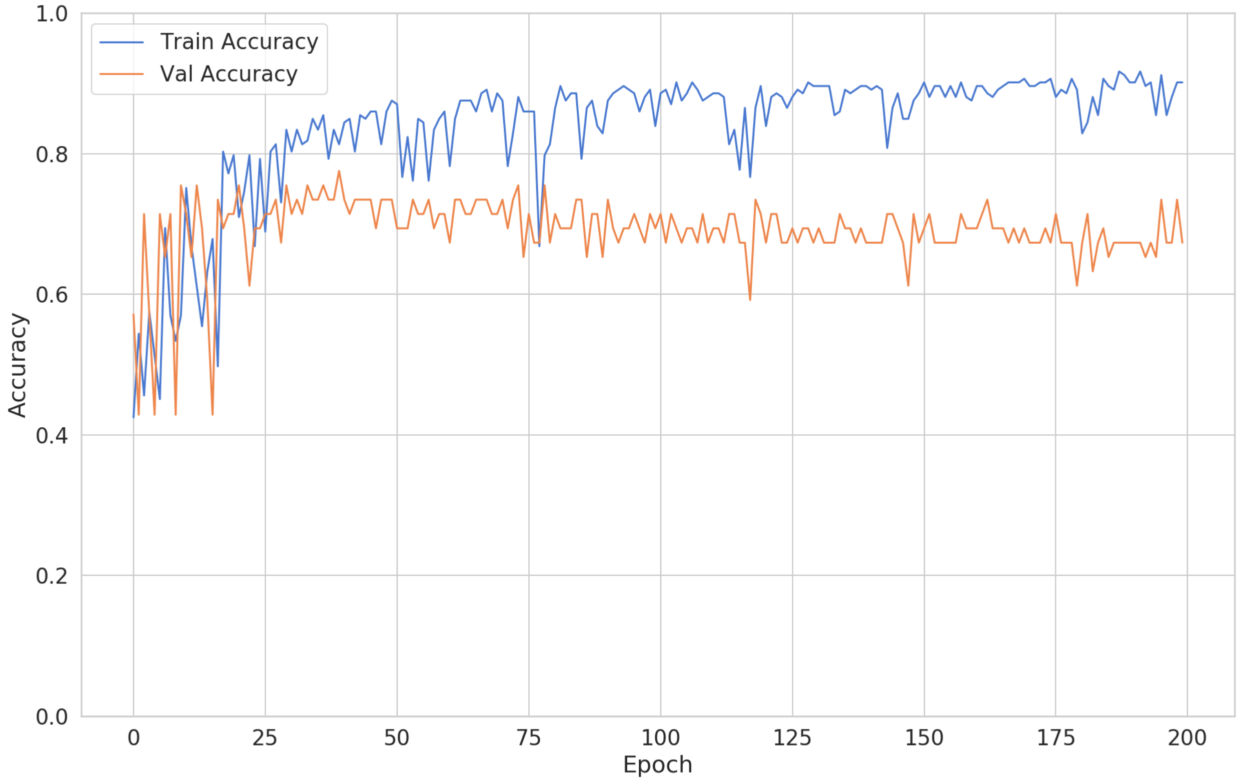
While this is an improvement, you can see that the validation loss starts to decrease after some time. How can you fix this?

Too complex model

We'll reuse the dataset but build a new model:

```
def build_classifier():
  model = keras.Sequential([
    keras.layers.Dense(
      units=128,
      activation='relu',
      input_shape=[X_train.shape[1]]
    ),
    keras.layers.Dense(units=64, activation='relu'),
    keras.layers.Dense(units=32, activation='relu'),
    keras.layers.Dense(units=16, activation='relu'),
    keras.layers.Dense(units=8, activation='relu'),
    keras.layers.Dense(units=1, activation='sigmoid'),
  ])

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

  return model
```

Here is the result:



You can see that the validation accuracy starts to decrease after epoch 25 or so.

The Fix #1

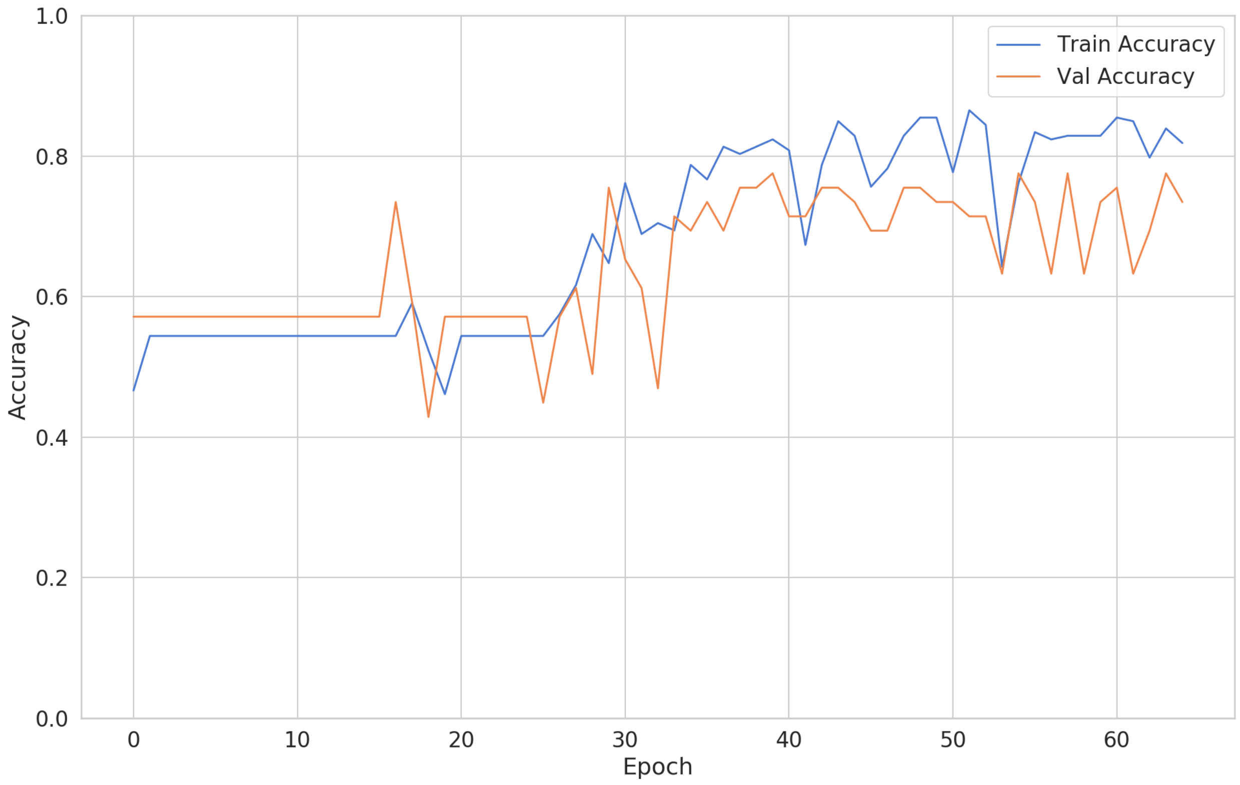
One way to fix this would be to simplify the model. But what if you spent so much time fine-tuning it? You can see that your model is performing better at a previous stage of the training.

You can use the EarlyStopping[152] callback to stop the training at some point:

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

```
clf = build_classifier()

early_stop = keras.callbacks.EarlyStopping(
  monitor='val_accuracy',
  patience=25
)

clf_history = clf.fit(
  x=X_train,
  y=y_train,
  shuffle=True,
  epochs=200,
  validation_split=0.2,
  batch_size=BATCH_SIZE,
  verbose=0,
  callbacks=[early_stop]
)
```

Here's the new training/validation loss:



Alright, looks like the training stopped much earlier than epoch 200. Faster training and a more accurate model. Nice!

The Fix #2

Another approach to fixing this problem is by using Regularization[153]. Regularization is a set of methods that forces the building of a less complex model. Usually, you get higher bias (less correct predictions on the training data) but reduced variance (higher accuracy on the validation dataset).

One of the most common ways to Regularize Neural Networks is by using Dropout[154].

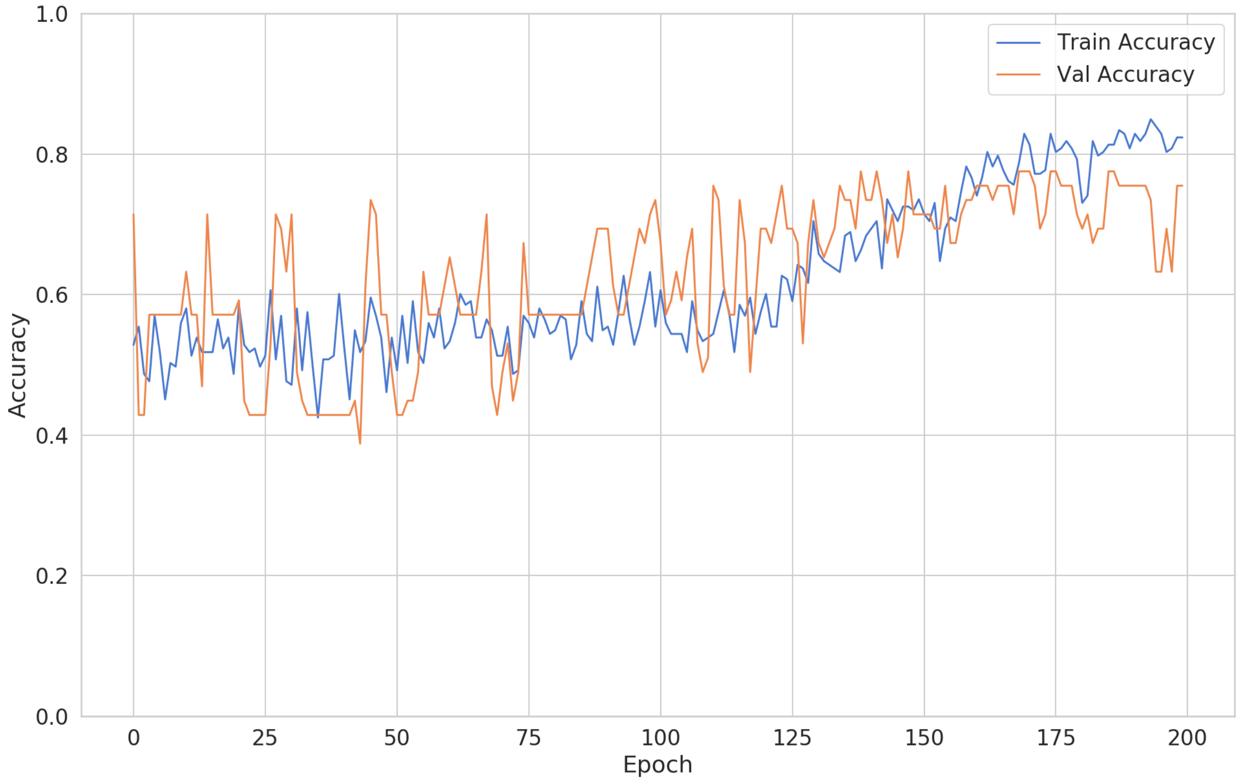
> Dropout is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks. The term "dropout" refers to dropping out units (both hidden and visible) in a neural network.

Using Dropout in Keras[155] is really easy:

```
model = keras.Sequential([
    keras.layers.Dense(
      units=128,
      activation='relu',
      input_shape=[X_train.shape[1]]
    ),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(units=64, activation='relu'),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(units=32, activation='relu'),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(units=16, activation='relu'),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(units=8, activation='relu'),
    keras.layers.Dense(units=1, activation='sigmoid'),
  ])

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

Here's how the training process has changed:

[153] https://en.wikipedia.org/wiki/Regularization_(mathematics)
[154] https://en.wikipedia.org/wiki/Dropout_(neural_networks)
[155] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout



The validation accuracy seems very good. Note that the training accuracy is down (we have a higher bias). There you have it, two ways to solve one issue!

Conclusion

Well done! You now have the toolset for dealing with the most common problems related to high bias or high variance. Here's a summary:

- Your data has no predictive power - use different data
- Your model is too simple to make good predictions - use model with more parameters
- Your data brings the Curse of dimensionality - use more data, reduce the number of features or use Bayesian Prior to provide more information
- Your model is too complex - use Early Stopping or Regularization to force creating a simpler model

Run the complete code in your browser[156]

[156] https://colab.research.google.com/drive/19wKH_-4srUuJDRiZIqpE06tfXF3MLp0i

References

- Bias-Variance Tradeoff in Machine Learning[157]
- Bias–variance tradeoff[158]
- Diagnosing Bias vs. Variance[159]
- What is the curse of dimensionality?[160]

[157] https://www.learnopencv.com/bias-variance-tradeoff-in-machine-learning/

[158] https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff

[159] https://www.coursera.org/lecture/machine-learning/diagnosing-bias-vs-variance-yCAup

[160] https://deepai.org/machine-learning-glossary-and-terms/curse-of-dimensionality

Hyperparameter Tuning

> TL;DR Learn how to search for good Hyperparameter values using Keras Tuner in your Keras and scikit-learn models

Hyperparameter tuning refers to the process of searching for the best subset of hyperparameter values in some predefined space. For us mere mortals, that means - should I use a learning rate of 0.001 or 0.0001?

In particular, tuning Deep Neural Networks is notoriously hard (that's what she said?). Choosing the number of layers, neurons, type of activation function(s), optimizer, and learning rate are just some of the options. Unfortunately, you don't really know which choices are the ones that matter, in advance.

On top of that, those models can be slow to train. Running many experiments in parallel might be a good option. Still, you need a lot of computational resources to do that on practical datasets.

Here are some of the ways that Hyperparameter tuning can help you:

- Better accuracy on the test set
- Reduced number of parameters
- Reduced number of layers
- Faster inference speed

None of these benefits are guaranteed, but in practice, some combination often is true.

Run the complete code in your browser[161]

What is a Hyperparameter?

Hyperparameters are never learned, but set by you (or your algorithm) and govern the whole training process. You can think of Hyperparameters as configuration variables you set when running some software. Common examples of Hyperparameters are learning rate, optimizer type, activation function, dropout rate.

Adjusting/finding good values is really slow. You have to wait for the whole training process to complete, evaluate the results and adjust the value(s). Unfortunately, you might have to complete the whole search process when your data or model changes.

Don't be a hero! Use Hyperparameters from papers or other peers when your datasets and models are similar. At least, you can use those as a starting point.

[161] https://colab.research.google.com/drive/1NnUdPslZubFyjek1dbzpIzi54jv0Cw0x

When to do Hyperparameter Tuning?

Changing anything inside your model or data affects the results from previous Hyperparameter searches. So, you want to defer the search as much as possible.

Three things need to be in place, before starting the search:

- You have intimate knowledge of your data
- You have an end-to-end framework/skeleton for running experiments
- You have a systematic way to record and check the results of the searches (coming up next)

Hyperparameter tuning can give you another 5-15% accuracy on the test data. Well worth it, if you have the computational resources to find a good set of parameters.

Common strategies

There are two common ways to search for hyperparameters:

Improving one model

This option suggest that you use a single model and try to improve it over time (days, weeks or even months). Each time you try to fiddle with the parameters so you get an improvement on your validation set.

This option is used when your dataset is very large and you lack computational resources to use the next one. (Grad student optimization also falls within this category)

Training many models

You train many models in parallel using different settings for the hyperparameters. This option is computationally demanding and can make your code messy.

Luckily, we'll use the Keras Tuner[162] to make the process more managable.

Finding Hyperparameters

We're searching for multiple parameters. It might sound tempting to try out every possible combination. Grid search is a good option for that.

[162] https://github.com/keras-team/keras-tuner

However, you might not want to do that. Random search is a better alternative[163]. It's just that Neural Networks seem much more sensitive to changes in one parameter than another.

Another approach is to use Bayesian Optimization[164]. This method builds a function that estimates how good your model is going to be with a certain choice of hyperparameters.

Both approaches are implemented in Keras Tuner. How can we use them?

> Remember to occasionaly re-evaluate your hyperparameters. Over time, you might've improved your algorithm, your dataset might have changed or the hardware/software has changed. Because of those changes the best settings for the hyperparameters can get stale and need to be re-evaluated.

Data

We'll use the Titanic survivor data from Kaggle[165]:

> The competition is simple: use machine learning to create a model that predicts which passengers survived the Titanic shipwreck.

Let's load and take a look at the training data:

```
!gdown --id 1uWHjZ3y9XZKpcJ4fkSwjQJ-VDbZS-7xi --output titanic.csv
```

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

Exploration

Let's take a quick look at the data and try to understand what it contains:

```
df.shape
```

```
(891, 12)
```

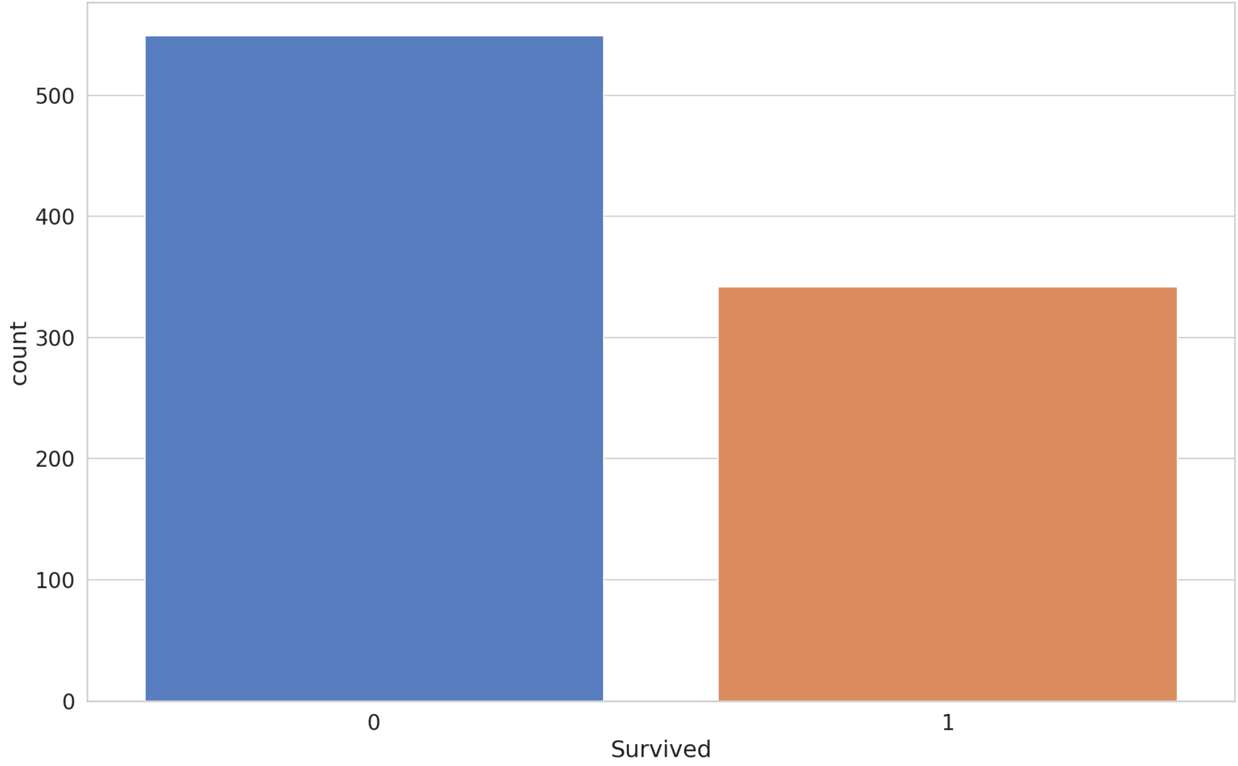
We have 12 columns with 891 rows. Let's see what the columns are:

[163] http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf
[164] https://arxiv.org/abs/1406.3896
[165] https://www.kaggle.com/c/titanic/data

```
df.columns
```

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

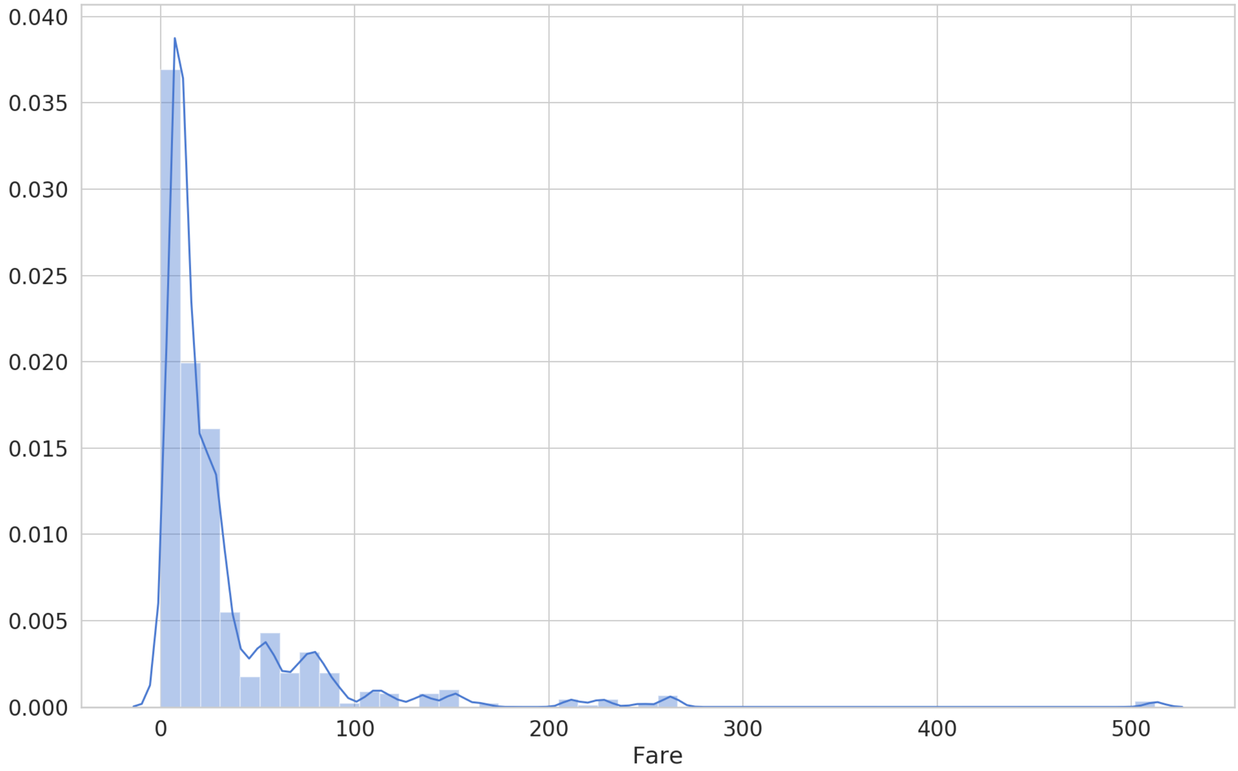
All of our models are going to predict the value of the `Survived` column. Let's have a look its distribution:



While the classes are not well balanced, we'll use the dataset as-is. Read the Practical Guide to Handling Imbalanced Datasets[166] to learn about some ways to solve this issue.

Another one that might interest you is the `Fare` (the price of the ticket):

[166] https://www.curiousily.com/posts/practical-guide-to-handling-imbalanced-datasets/



About 80% of the tickets are priced below 30 USD. Do we have missing data?

Preprocessing

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

```
Cabin       687
Age         177
Embarked      2
```

Yes, we have a lot of cabin data missing. Luckily, we won't need that feature for our model. Let's drop it along with other columns:

```
df = df.drop(['Cabin', 'Name', 'Ticket', 'PassengerId'], axis=1)
```

We're left with 8 columns (including `Survived`). We still have to do something with the missing `Age` and `Embarked` columns. Let's handle those:

```
df['Age'] = df['Age'].fillna(df['Age'].mean())
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].mode()[0])
```

The missing `Age` values are replaced with the mean value. Missing `Embarked` values are replaced with the most common one.

Now that our dataset has no missing values, we need preprocess the categorical features:

```
df = pd.get_dummies(df, columns=['Sex', 'Embarked', 'Pclass'])
```

We can start with building and optimizing our models. What do we need?

Keras Tuner

Keras Tuner[167] is a new library (still in beta) that promises:

> Hyperparameter tuning for humans

Sounds cool. Let's have a closer look.

There are two main requirements for searching Hyperparameters with Keras Tuner:

- Create a model building function that specifies possible Hyperparameter values
- Create and configure a Tuner to use for the search process

The version of Keras Tuner we're using in this writing is 7f6b00f45c6e0b0debaf183fa5f9dcef824fb02f[168]. Yes, we're using the code from the `master` branch.

There are four different tuners available:

- RandomSearch[169]
- Hyperband[170]
- BayesianOptimization[171]
- Sklearn[172]

The scikit-learn Tuner is a bit special. It doesn't implement any algorithm for searching Hyperparameters. It rather relies on existing strategies to tune scikit-learn models.

How can we use Keras Tuner to find good parameters?

Random Search

Let's start with a complete example of how we can tune a model using Random Search:

[167] https://github.com/keras-team/keras-tuner
[168] https://github.com/keras-team/keras-tuner/commit/7f6b00f45c6e0b0debaf183fa5f9dcef824fb02f
[169] https://github.com/keras-team/keras-tuner/blob/master/kerastuner/tuners/randomsearch.py
[170] https://github.com/keras-team/keras-tuner/blob/master/kerastuner/tuners/hyperband.py
[171] https://github.com/keras-team/keras-tuner/blob/master/kerastuner/tuners/bayesian.py
[172] https://github.com/keras-team/keras-tuner/blob/master/kerastuner/tuners/sklearn.py

```
def tune_optimizer_model(hp):
    model = keras.Sequential()
    model.add(keras.layers.Dense(
      units=18,
      activation="relu",
      input_shape=[X_train.shape[1]]
    ))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    optimizer = hp.Choice('optimizer', ['adam', 'sgd', 'rmsprop'])

    model.compile(
        optimizer=optimizer,
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Everything here should look familiar except for the way we're choosing an Optimizer. We register a Hyperparameter with the name of `optimizer` and the available options. The next step is to create a Tuner:

```
MAX_TRIALS = 20
EXECUTIONS_PER_TRIAL = 5

tuner = RandomSearch(
    tune_optimizer_model,
    objective='val_accuracy',
    max_trials=MAX_TRIALS,
    executions_per_trial=EXECUTIONS_PER_TRIAL,
    directory='test_dir',
    project_name='tune_optimizer',
    seed=RANDOM_SEED
)
```

The Tuner needs a pointer to the model building function, what objective should optimize for (validation accuracy), and how many model configurations to test at most. The other config settings are rather self-explanatory.

We can get a summary of the different parameter values from our Tuner:

```
tuner.search_space_summary()
```

```
Search space summary
|-Default search space size: 1
optimizer (Choice)
|-default: adam
|-ordered: False
|-values: ['adam', 'sgd', 'rmsprop']
```

Finally, we can start the search:

```
TRAIN_EPOCHS = 20

tuner.search(x=X_train,
             y=y_train,
             epochs=TRAIN_EPOCHS,
             validation_data=(X_test, y_test))
```

The search process saves the trials for later analysis/reuse. Keras Tunes makes it easy to obtain previous results and load the best model found so far.

You can get a summary of the results:

```
tuner.results_summary()
```

```
Results summary
|-Results in test_dir/tune_optimizer
|-Showing 10 best trials
|-Objective: Objective(name='val_accuracy', direction='max') Score: 0.75195533037185\
67
|-Objective: Objective(name='val_accuracy', direction='max') Score: 0.74301671981811\
52
|-Objective: Objective(name='val_accuracy', direction='max') Score: 0.72737431526184\
08
```

That's not helpful since we can't get the actual values of the Hyperparameters. Follow this issue[173] for resolution of this.

Luckily, we can obtain the Hyperparameter values like so:

[173] https://github.com/keras-team/keras-tuner/issues/121

```
tuner.oracle.get_best_trials(num_trials=1)[0].hyperparameters.values
```

```
{'optimizer': 'adam'}
```

Even better, we can get the best performing model:

```
best_model = tuner.get_best_models()[0]
```

Ok, choosing an Optimizer looks easy enough. What else can we tune?

Learning rate and Momentum

The following examples use the same RandomSearch settings. We'll change the model building function.

Two of the most important parameters for your Optimizer are the Learning rate[174] and Momentum[175]. Let's try to find good values for those:

```
def tune_rl_momentum_model(hp):
    model = keras.Sequential()
    model.add(keras.layers.Dense(
      units=18,
      activation="relu",
      input_shape=[X_train.shape[1]]
    ))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    lr = hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])
    momentum = hp.Choice('momentum', [0.0, 0.2, 0.4, 0.6, 0.8, 0.9])

    model.compile(
        optimizer=keras.optimizers.SGD(lr, momentum=momentum),
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

The procedure is pretty identical to the one we've used before. Here are the results:

[174] https://en.wikipedia.org/wiki/Learning_rate
[175] https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Momentum

```
{'learning_rate': 0.01, 'momentum': 0.4}
```

Number of parameters

We can also try to find better value for the number of units in our hidden layer:

```
def tune_neurons_model(hp):
    model = keras.Sequential()
    model.add(keras.layers.Dense(units=hp.Int('units',
                                        min_value=8,
                                        max_value=128,
                                        step=16),
                                 activation="relu",
                                 input_shape=[X_train.shape[1]]))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

We're using a range of values for the number of parameters. The range is defined by a minimum, maximum and step value. The best number of units is:

```
{'units': 72}
```

Number of hidden layers

We can use Hyperparameter tuning for finding a better architecture for our model. Keras Tuner allows us to use regular Python for loops to do that:

```python
def tune_layers_model(hp):
    model = keras.Sequential()

    model.add(keras.layers.Dense(units=128,
                                 activation="relu",
                                 input_shape=[X_train.shape[1]]))

    for i in range(hp.Int('num_layers', 1, 6)):
      model.add(keras.layers.Dense(units=hp.Int('units_' + str(i),
                                              min_value=8,
                                              max_value=64,
                                              step=8),
                               activation='relu'))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Note that we still test a different number of units for each layer. There is a requirement that each Hyperparameter name should be unique. We get:

```
{'num_layers': 2,
 'units_0': 32,
 'units_1': 24,
 'units_2': 64,
 'units_3': 8,
 'units_4': 48,
 'units_5': 64}
```

Not that informative. Well, you can still get the best model and run with it.

Activation function

You can try out different activation functions like so:

```python
def tune_act_model(hp):
    model = keras.Sequential()

    activation = hp.Choice('activation',
                           [
                             'softmax',
                             'softplus',
                             'softsign',
                             'relu',
                             'tanh',
                             'sigmoid',
                             'hard_sigmoid',
                             'linear'
                           ])

    model.add(keras.layers.Dense(units=32,
                                 activation=activation,
                                 input_shape=[X_train.shape[1]]))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Surprisingly we obtain the following result:

```
{'activation': 'linear'}
```

Dropout rate

Dropout[176] is a frequently used Regularization technique. Let’s try different rates:

[176] http://jmlr.org/papers/v15/srivastava14a.html

```
def tune_dropout_model(hp):
    model = keras.Sequential()

    drop_rate = hp.Choice('drop_rate',
                            [
                              0.0,
                              0.1,
                              0.2,
                              0.3,
                              0.4,
                              0.5,
                              0.6,
                              0.7,
                              0.8,
                              0.9
                            ])

    model.add(keras.layers.Dense(units=32,
                                 activation="relu",
                                 input_shape=[X_train.shape[1]]))
    model.add(keras.layers.Dropout(rate=drop_rate))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Unsurprisingly, our model is relatively small and don't benefit from regularization:

```
{'drop_rate': 0.0}
```

Complete example

We've dabbled with the Keras Tuner API for a bit. Let's have a look at a somewhat more realistic example:

```
def tune_nn_model(hp):
    model = keras.Sequential()

    model.add(keras.layers.Dense(units=128,
                                 activation="relu",
                                 input_shape=[X_train.shape[1]]))

    for i in range(hp.Int('num_layers', 1, 6)):
      units = hp.Int(
          'units_' + str(i),
          min_value=8,
          max_value=64,
          step=8
      )
      model.add(keras.layers.Dense(units=units, activation='relu'))
      drop_rate = hp.Choice('drop_rate_' + str(i),
                            [
                              0.0, 0.1, 0.2, 0.3, 0.4,
                              0.5, 0.6, 0.7, 0.8, 0.9
                            ])
      model.add(keras.layers.Dropout(rate=drop_rate))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Yes, tuning parameters can complicate your code. One thing that might be helpful is to try and separate the possible Hyperparameter values from the code building code.

Bayesian Optimization

The Bayesian Tuner provides the same API as Random Search. In practice, this method should be as good (if not better) as the Grad student hyperparameter tuning method. Let's have a look:

```
b_tuner = BayesianOptimization(
    tune_nn_model,
    objective='val_accuracy',
    max_trials=MAX_TRIALS,
    executions_per_trial=EXECUTIONS_PER_TRIAL,
    directory='test_dir',
    project_name='b_tune_nn',
    seed=RANDOM_SEED
)
```

This method might try out significantly fewer parameters than Random Search, but this is highly problem dependent. I would recommend using this Tuner for most practical problems.

scikit-learn model tuning

Despite its name, Keras Tuner allows you to tune scikit-learn models too! Let's try it out on a RandomForestClassifier[177]:

```
import kerastuner as kt
from sklearn import ensemble
from sklearn import metrics
from sklearn import datasets
from sklearn import model_selection

def build_tree_model(hp):
  return ensemble.RandomForestClassifier(
      n_estimators=hp.Int('n_estimators', 10, 80, step=5),
      max_depth=hp.Int('max_depth', 3, 10, step=1),
      max_features=hp.Choice('max_features', ['auto', 'sqrt', 'log2'])
  )
```

We'll tune the number of trees in the forest (**n_estimators**), the maximum depth of the trees (**max_depth**), and the number of features to consider when choosing the best split (**max_features**).

The Tuner expects an optimization strategy (Oracle). We'll use Baysian Optimization:

[177] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

```
sk_tuner = kt.tuners.Sklearn(
  oracle=kt.oracles.BayesianOptimization(
      objective=kt.Objective('score', 'max'),
      max_trials=MAX_TRIALS,
      seed=RANDOM_SEED
  ),
  hypermodel=build_tree_model,
  scoring=metrics.make_scorer(metrics.accuracy_score),
  cv=model_selection.StratifiedKFold(5),
  directory='test_dir',
  project_name='tune_rf'
)
```

The rest of the API is identical:

```
sk_tuner.search(X_train.values, y_train.values)
```

The best parameter values are:

```
sk_tuner.oracle.get_best_trials(num_trials=1)[0].hyperparameters.values
```

```
{'max_depth': 4, 'max_features': 'sqrt', 'n_estimators': 60}
```

Conclusion

There you have it. You now know how to search for good Hyperparameters for Keras and scikit-learn models.

Remember the three requirements that need to be in place before starting the search:

- You have intimate knowledge of your data
- You have an end-to-end framework/skeleton for running experiments
- You have a systematic way to record and check the results of the searches

Keras Tuner can help you with the last step.

Run the complete code in your browser[178]

[178] https://colab.research.google.com/drive/1NnUdPslZubFyjek1dbzpIzi54jv0Cw0x

References

- Keras Tuner[179]
- Random Search for Hyper-Parameter Optimization[180]
- Bayesian optimization[181]
- Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization[182]
- Overview of hyperparameter tuning[183]

[179] https://github.com/keras-team/keras-tuner
[180] http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf
[181] https://krasserm.github.io/2018/03/21/bayesian-optimization/
[182] https://arxiv.org/abs/1603.06560
[183] https://cloud.google.com/ml-engine/docs/hyperparameter-tuning-overview

Heart Disease Prediction

> TL;DR Build and train a Deep Neural Network for binary classification in TensorFlow 2. Use the model to predict the presence of heart disease from patient data.

Machine Learning is used to solve real-world problems in many areas, already. Medicine is no exception. While controversial, multiple models have been proposed and used with some success. Some notable projects by Google and others:

- Diagnosing Diabetic Eye Disease[184]
- Assisting Pathologists in Detecting Cancer[185]

Today, we're going to take a look at one specific area - heart disease prediction.

> About 610,000 people die of heart disease in the United States every year – that's 1 in every 4 deaths. Heart disease is the leading cause of death for both men and women. More than half of the deaths due to heart disease in 2009 were in men. - Heart Disease Facts & Statistics | cdc.gov[186]

Please note, the model presented here is very limited and in no way applicable for real-world situations. Our dataset is extremely small, conclusions made here are in no way generalizable. Heart disease prediction is a vastly more complex problem than depicted in this writing.

Complete source code in Google Colaboratory Notebook[187]

Here is the plan:

1. Explore patient data
2. Data preprocessing
3. Create your Neural Network in TensorFlow 2
4. Train the model
5. Predict heart disease from patient data

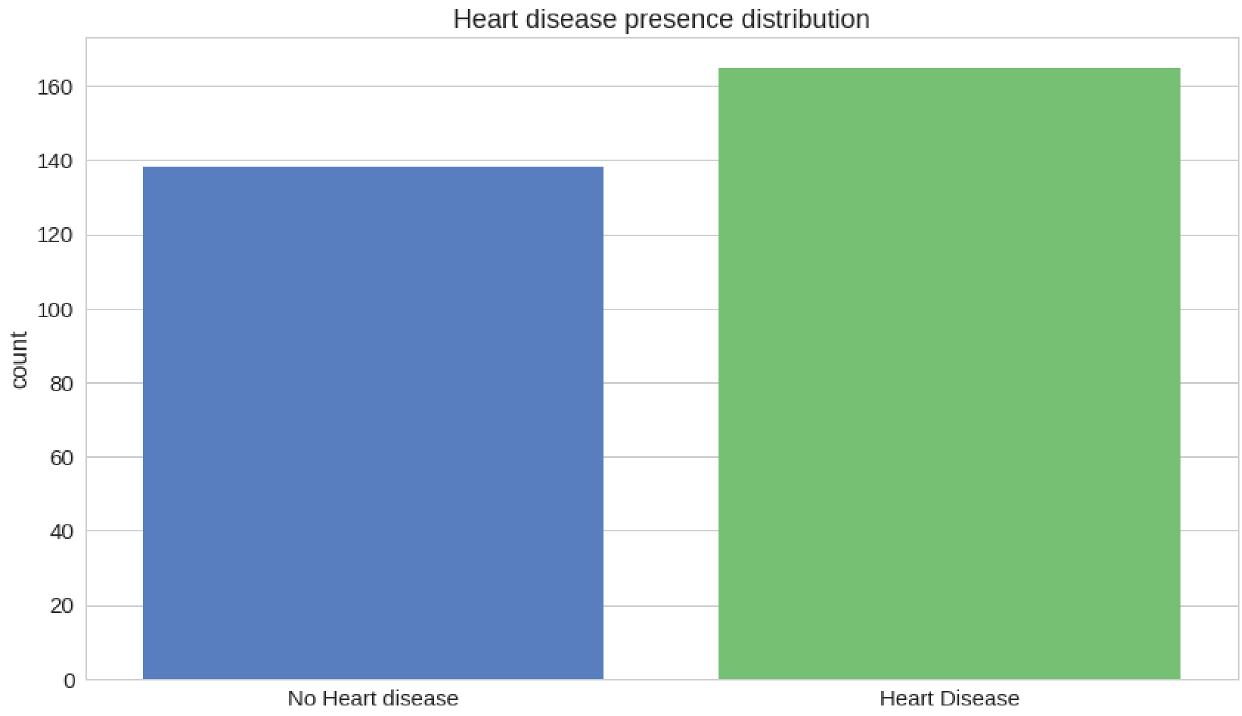
Patient Data

Our data comes from this dataset[188]. It contains *303* patient records. Each record contains *14* attributes:

[184] https://ai.googleblog.com/2016/11/deep-learning-for-detection-of-diabetic.html
[185] https://ai.googleblog.com/2017/03/assisting-pathologists-in-detecting.html
[186] https://www.cdc.gov/heartdisease/facts.htm
[187] https://colab.research.google.com/drive/13EThgYKSRwGBJJn_8iAvg-QWUWjCufB1
[188] https://www.kaggle.com/ronitf/heart-disease-uci

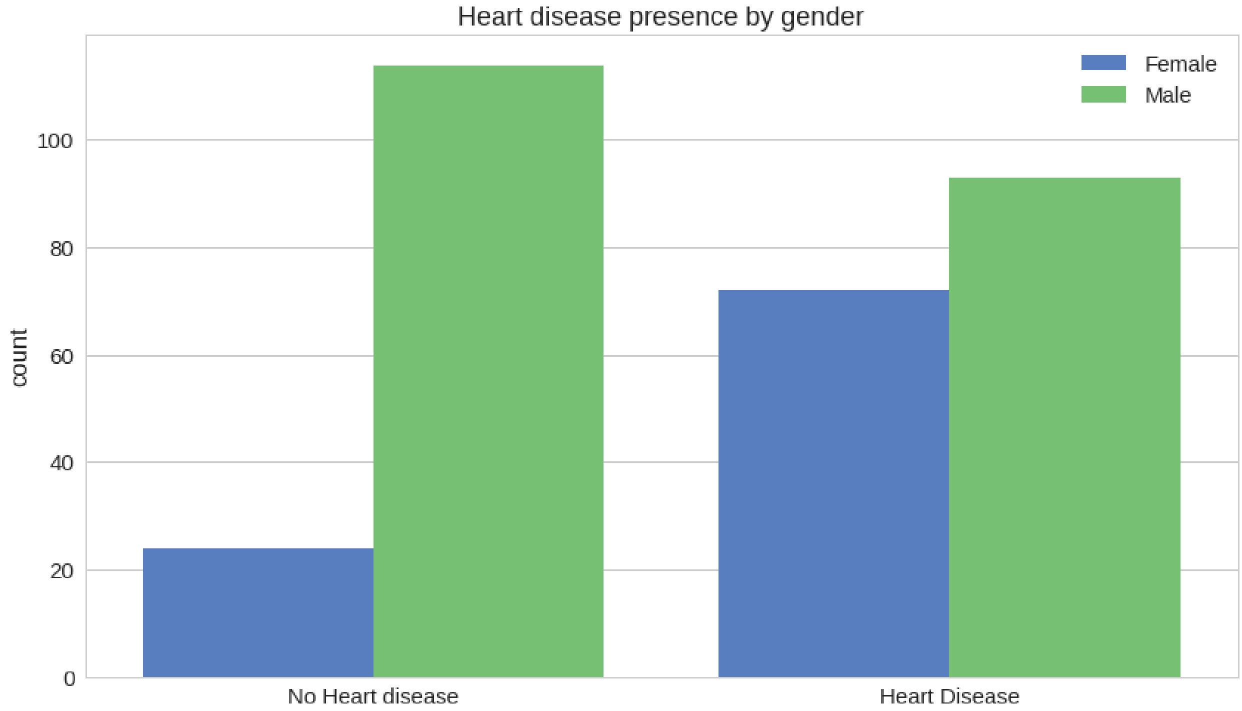
Label	Description
age	age in years
sex	(1 = male; 0 = female)
cp	(1 = typical angina; 2 = atypical angina; 3 = non-anginal pain; 4 = asymptomatic)
trestbps	resting blood pressure (in mm Hg on admission to the hospital)
chol	serum cholestoral in mg/dl
fbs	(fasting blood sugar > 120 mg/dl) (1 = true; 0 = false)
restecg	resting electrocardiographic results
thalach	maximum heart rate achieved
exang	exercise induced angina (1 = yes; 0 = no)
oldpeak	ST depression induced by exercise relative to rest
slope	the slope of the peak exercise ST segment
ca	number of major vessels (0-3) colored by flourosopy
thal	(3 = normal; 6 = fixed defect; 7 = reversable defect)
target	(0 = no heart disease; 1 = heart disease presence)

How many of the patient records indicate heart disease?

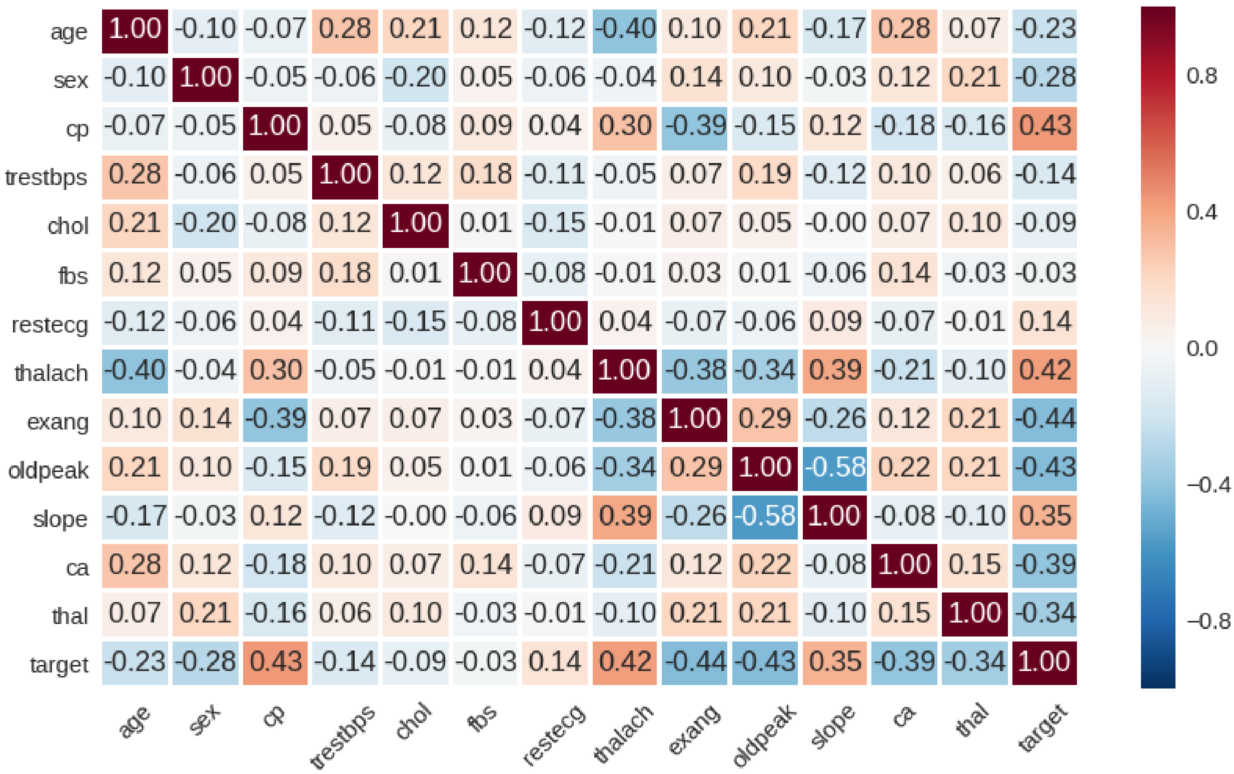


That looks like a pretty well-distributed dataset, considering the number of rows.

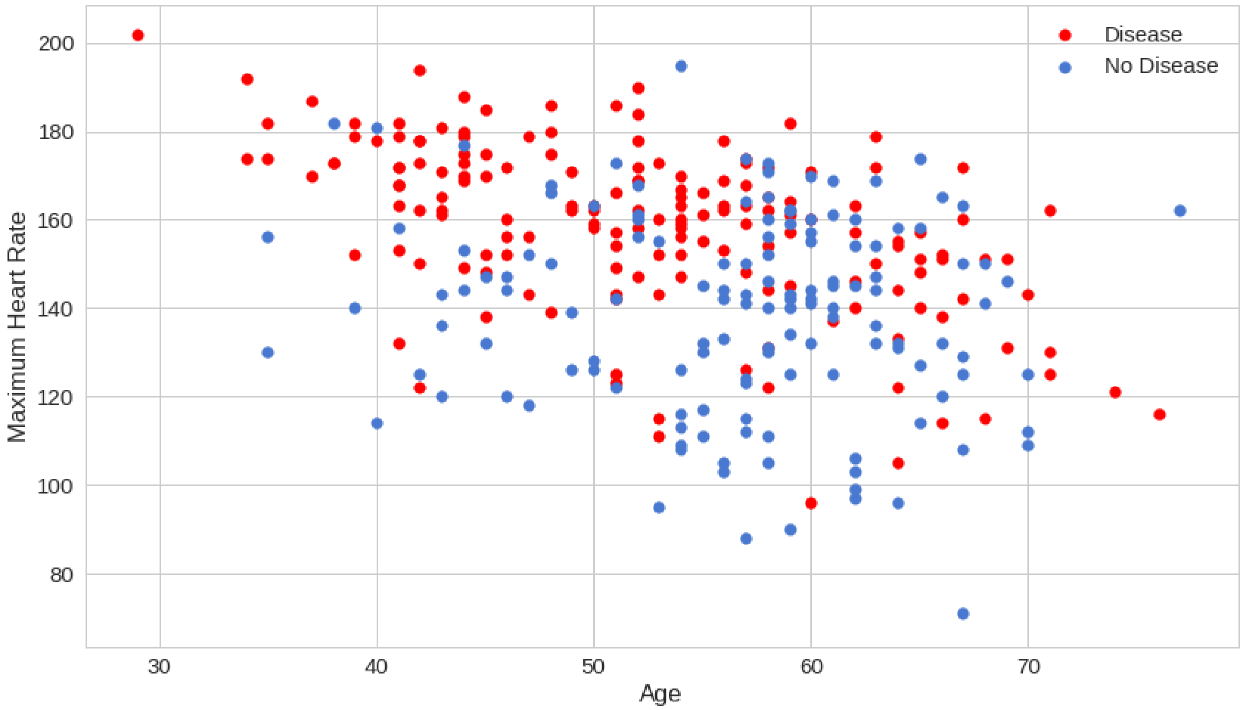
Let’s have a look at how heart disease affects different genders:



Here is a Pearson correlation heatmap between the features:

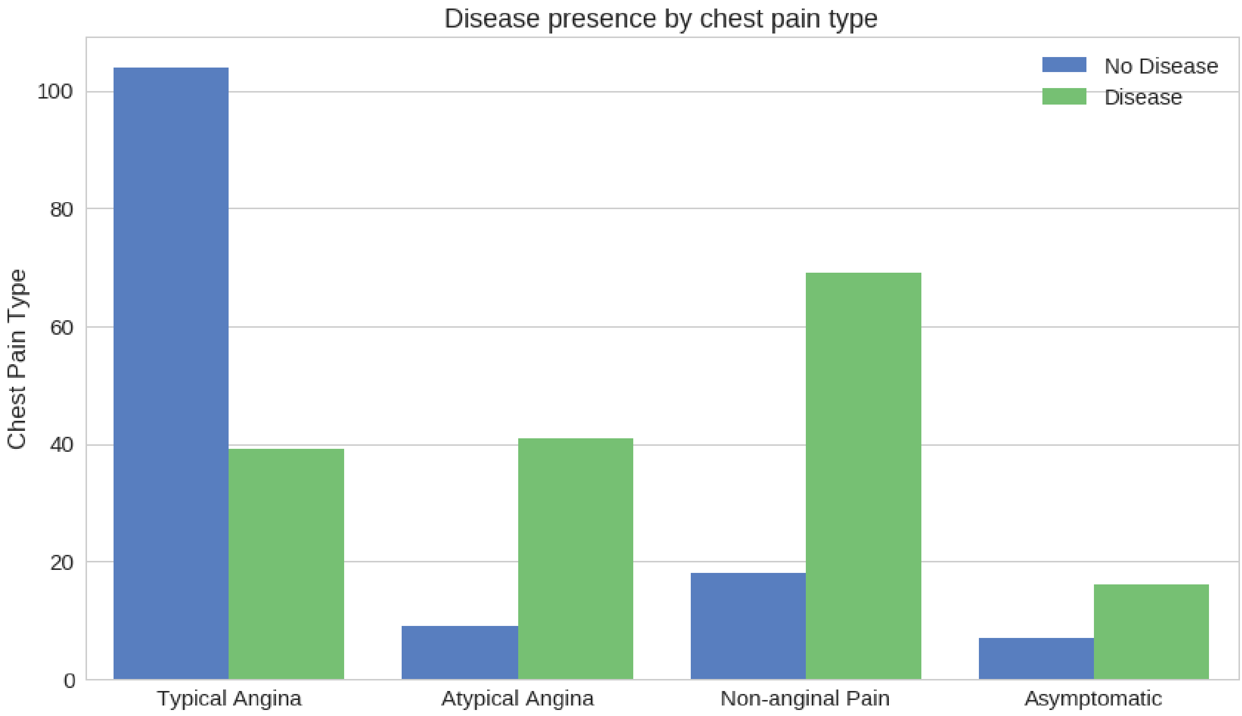


How disease presence is affected by `thalach` ("Maximum Heart Rate") vs `age`:



Looks like maximum heart rate can be very predictive for the presence of a disease, regardless of age.

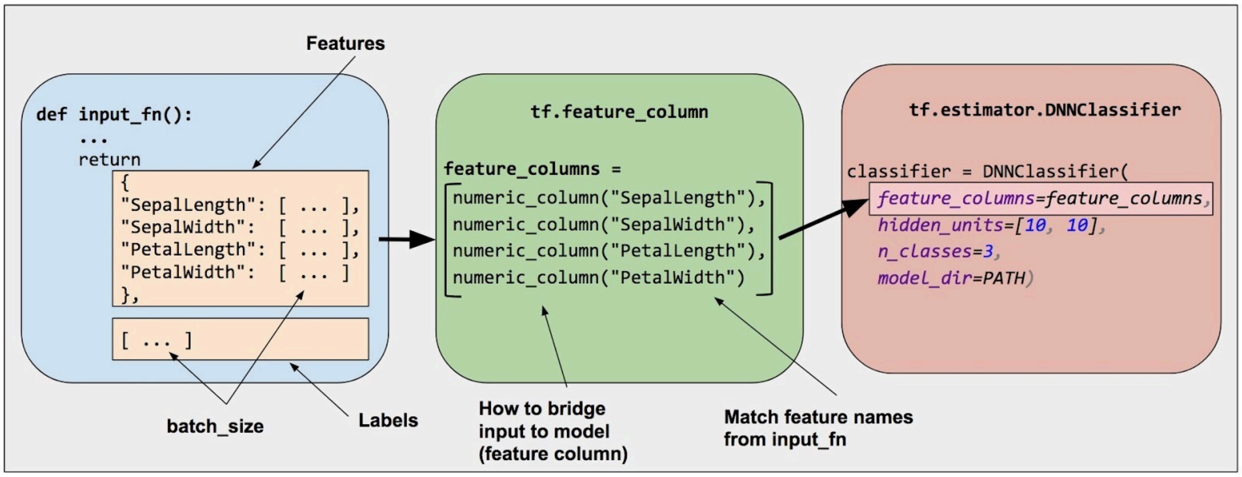
How different types of chest pain affect the presence of heart disease:



Having chest pain might not be indicative of heart disease.

Data Preprocessing

Our data contains a mixture of categorical and numerical data. Let's use TensorFlow's Feature Columns[189].



```
def input_fn():
    ...
    return
        {
        "SepalLength": [ ... ],
        "SepalWidth":  [ ... ],
        "PetalLength": [ ... ],
        "PetalWidth":  [ ... ]
        },
        [ ... ]
```

```
feature_columns =
[numeric_column("SepalLength"),
 numeric_column("SepalWidth"),
 numeric_column("PetalLength"),
 numeric_column("PetalWidth")
]
```

```
classifier = DNNClassifier(
  feature_columns=feature_columns,
  hidden_units=[10, 10],
  n_classes=3,
  model_dir=PATH)
```

[189] https://www.tensorflow.org/guide/feature_columns

Feature columns allow you to bridge/process the raw data in your dataset to fit your model input data requirements. Furthermore, you can separate the model building process from the data preprocessing. Let's have a look:

```python
feature_columns = []

## numeric cols
for header in ['age', 'trestbps', 'chol', 'thalach', 'oldpeak', 'ca']:
  feature_columns.append(tf.feature_column.numeric_column(header))

## bucketized cols
age = tf.feature_column.numeric_column("age")
age_buckets = tf.feature_column.bucketized_column(age, boundaries=[18, 25, 30, 35, 4\
0, 45, 50, 55, 60, 65])
feature_columns.append(age_buckets)

## indicator cols
data["thal"] = data["thal"].apply(str)
thal = tf.feature_column.categorical_column_with_vocabulary_list(
      'thal', ['3', '6', '7'])
thal_one_hot = tf.feature_column.indicator_column(thal)
feature_columns.append(thal_one_hot)

data["sex"] = data["sex"].apply(str)
sex = tf.feature_column.categorical_column_with_vocabulary_list(
      'sex', ['0', '1'])
sex_one_hot = tf.feature_column.indicator_column(sex)
feature_columns.append(sex_one_hot)

data["cp"] = data["cp"].apply(str)
cp = tf.feature_column.categorical_column_with_vocabulary_list(
      'cp', ['0', '1', '2', '3'])
cp_one_hot = tf.feature_column.indicator_column(cp)
feature_columns.append(cp_one_hot)

data["slope"] = data["slope"].apply(str)
slope = tf.feature_column.categorical_column_with_vocabulary_list(
      'slope', ['0', '1', '2'])
slope_one_hot = tf.feature_column.indicator_column(slope)
feature_columns.append(slope_one_hot)
```

Apart from the numerical features, we're putting patient `age` into discrete ranges (buckets). Furthermore, `thal`, `sex`, `cp`, and `slope` are categorical and we map them to such.

Next up, lets turn the pandas DataFrame into a TensorFlow Dataset:

```
def create_dataset(dataframe, batch_size=32):
  dataframe = dataframe.copy()
  labels = dataframe.pop('target')
  return tf.data.Dataset.from_tensor_slices((dict(dataframe), labels)) \
          .shuffle(buffer_size=len(dataframe)) \
          .batch(batch_size)
```

And split the data into training and testing:

```
train, test = train_test_split(
  data,
  test_size=0.2,
  random_state=RANDOM_SEED
)

train_ds = create_dataset(train)
test_ds = create_dataset(test)
```

The Model

Let's build a binary classifier using Deep Neural Network in TensorFlow:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.DenseFeatures(feature_columns=feature_columns),
  tf.keras.layers.Dense(units=128, activation='relu'),
  tf.keras.layers.Dropout(rate=0.2),
  tf.keras.layers.Dense(units=128, activation='relu'),
  tf.keras.layers.Dense(units=2, activation='sigmoid')
])
```

Our model uses the feature columns we've created in the preprocessing step. Note that, we're no longer required to specify the input layer size.

We also use the Dropout[190] layer between 2 dense layers. Our output layer contains 2 neurons, since we are building a binary classifier.

[190] https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Dropout

Training

Our loss function is binary cross-entropy defined by:

$$-(y\log(p) + (1 - y)\log(1 - p))$$

where y is binary indicator if the predicted class is correct for the current observation and p is the predicted probability.

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

history = model.fit(
  train_ds,
  validation_data=test_ds,
  epochs=100,
  use_multiprocessing=True
)
```

Here is a sample of the training process:

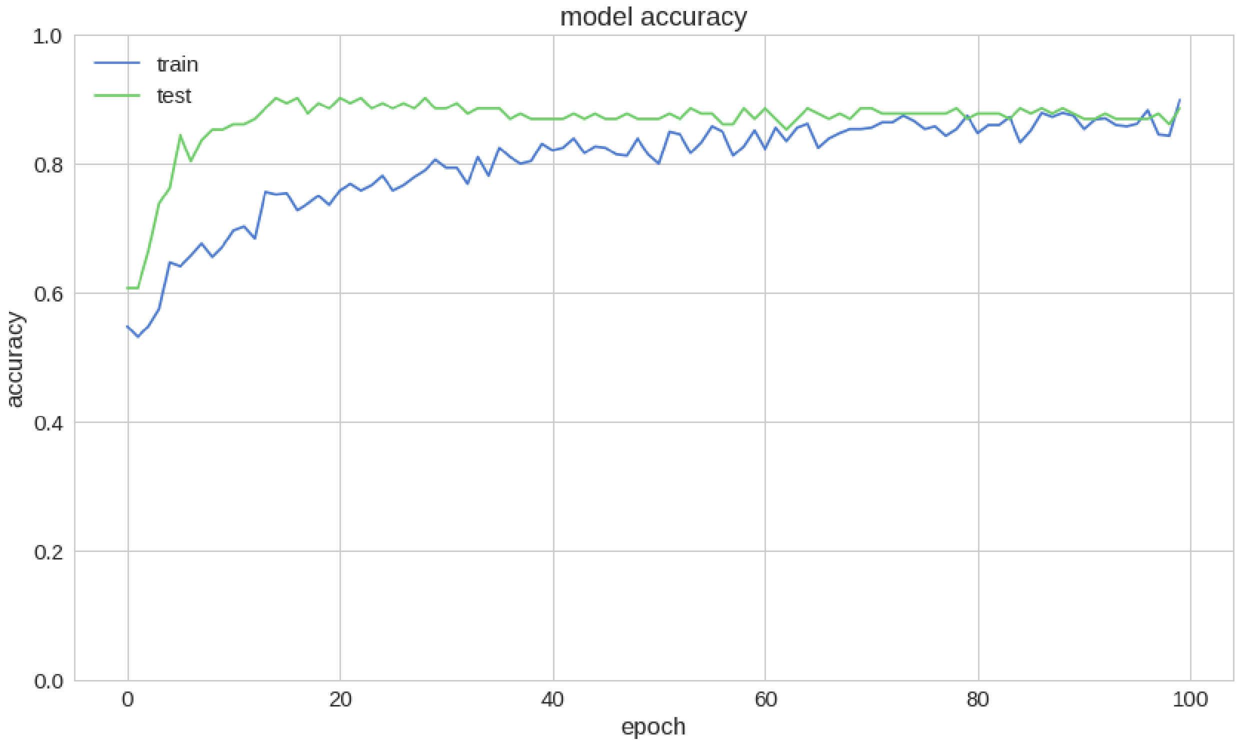
```
Epoch 95/100
0s 42ms/step - loss: 0.3018 - accuracy: 0.8430 - val_loss: 0.4012 - val_accuracy: 0.\
8689
Epoch 96/100
0s 42ms/step - loss: 0.2882 - accuracy: 0.8547 - val_loss: 0.3436 - val_accuracy: 0.\
8689
Epoch 97/100
0s 42ms/step - loss: 0.2889 - accuracy: 0.8732 - val_loss: 0.3368 - val_accuracy: 0.\
8689
Epoch 98/100
0s 42ms/step - loss: 0.2964 - accuracy: 0.8386 - val_loss: 0.3537 - val_accuracy: 0.\
8770
Epoch 99/100
0s 43ms/step - loss: 0.3062 - accuracy: 0.8282 - val_loss: 0.4110 - val_accuracy: 0.\
8607
Epoch 100/100
0s 43ms/step - loss: 0.2685 - accuracy: 0.8821 - val_loss: 0.3669 - val_accuracy: 0.\
8852
```

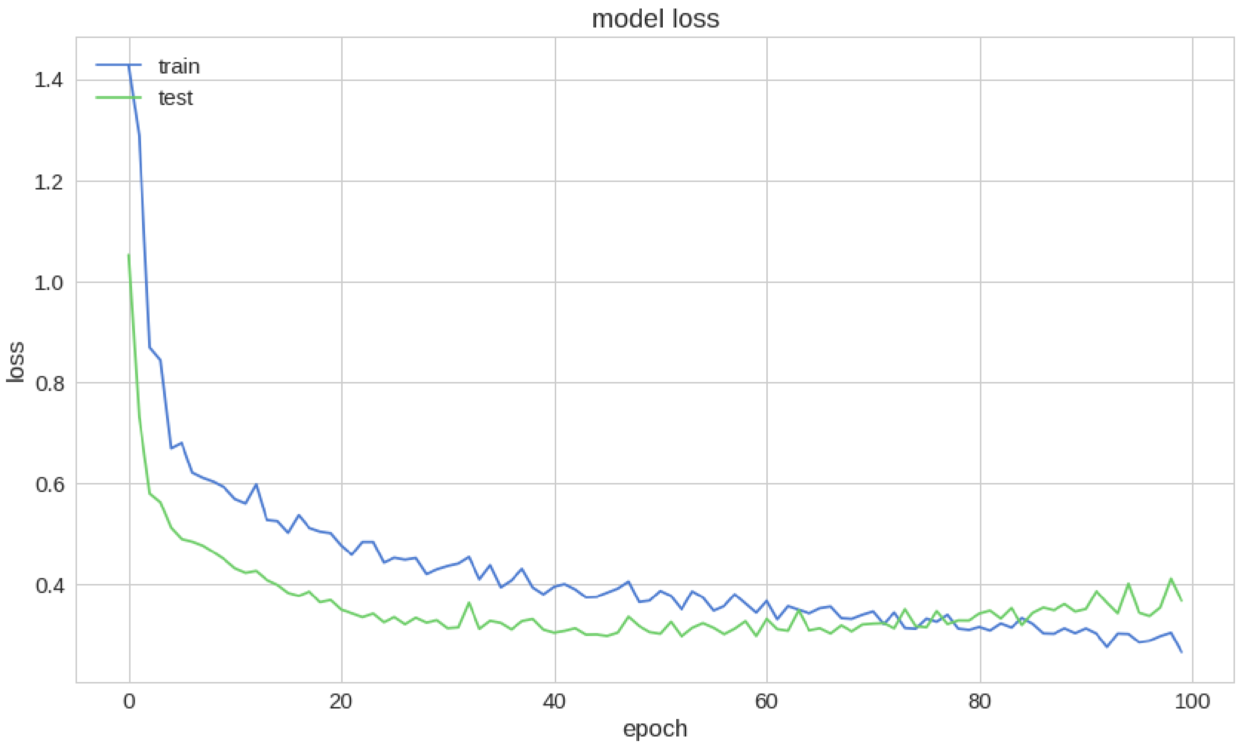
Accuracy on the test set:

```
model.evaluate(test_ds)
```

```
0s 24ms/step - loss: 0.3669 - accuracy: 0.8852
[0.3669000566005707, 0.8852459]
```

So, we have ~*88%* accuracy on the test set.





Predicting Heart Disease

Now that we have a model with some good accuracy on the test set, let's try to predict heart disease based on the features in our dataset.

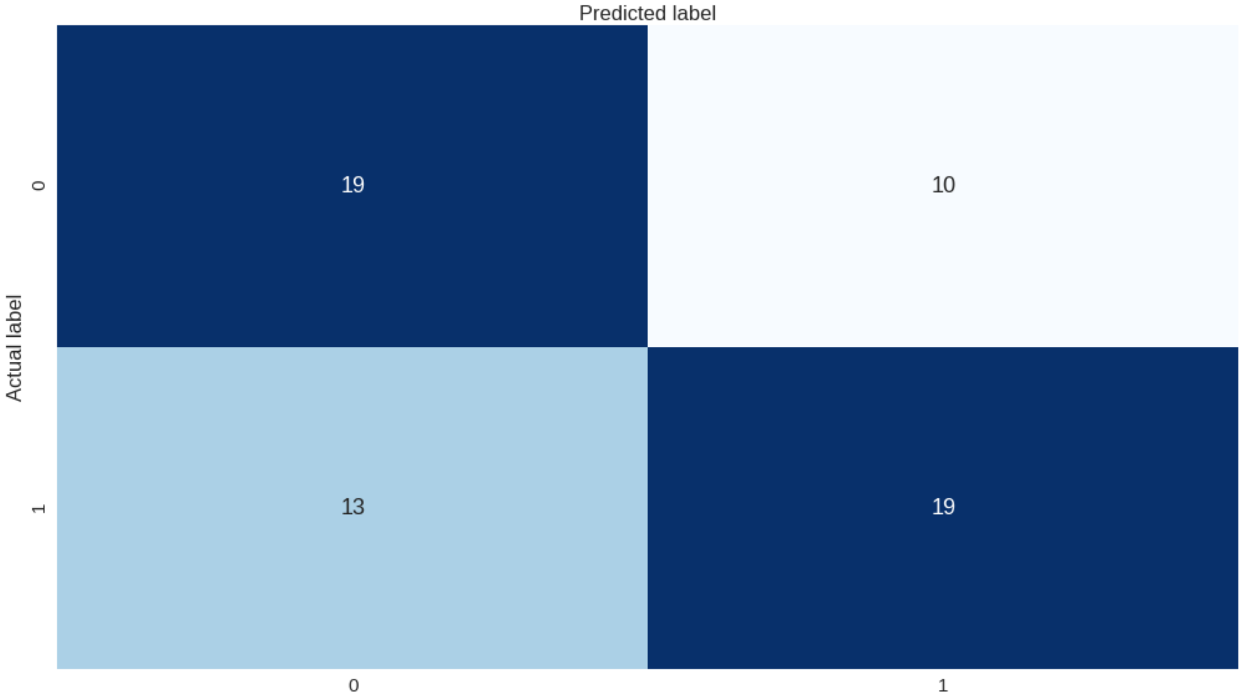
```
predictions = tf.round(model.predict(test_ds)).numpy().flatten()
```

Since we're interested in making binary decisions, we're taking the maximum probability of the output layer.

```
print(classification_report(y_test.values, predictions))
```

```
              precision    recall  f1-score   support

           0       0.59      0.66      0.62        29
           1       0.66      0.59      0.62        32

   micro avg       0.62      0.62      0.62        61
   macro avg       0.62      0.62      0.62        61
weighted avg       0.63      0.62      0.62        61
```

Regardless of the accuracy, you can see that the precision, recall and f1-score of our model are not that high. Let's take a look at the confusion matrix:



Our model looks a bit confused. Can you improve on it?

Conclusion

Complete source code in Google Colaboratory Notebook[191]

You did it! You made a binary classifier using Deep Neural Network with TensorFlow and used it to predict heart disease from patient data.

Next, we'll have a look at what TensorFlow 2 has in store for us, when applied to computer vision.

[191] https://colab.research.google.com/drive/13EThgYKSRwGBJJn_8iAvg-QWUWjCufB1

Time Series Forecasting

> TL;DR Learn about Time Series and making predictions using Recurrent Neural Networks. Prepare sequence data and use LSTMs to make simple predictions.

Often you might have to deal with data that does have a time component. No matter how much you squint your eyes, it will be difficult to make your favorite data independence assumption. It seems like newer values in your data might depend on the historical values. How can you use that kind of data to build models?

This guide will help you better understand Time Series data and how to build models using Deep Learning (Recurrent Neural Networks). You'll learn how to preprocess Time Series, build a simple LSTM model, train it, and use it to make predictions. Here are the steps:

- Time Series
- Recurrent Neural Networks
- Time Series Prediction with LSTMs

Run the complete notebook in your browser[192]

The complete project on GitHub[193]

Time Series

Time Series[194] is a collection of data points indexed based on the time they were collected. Most often, the data is recorded at regular time intervals. What makes Time Series data special?

Forecasting future Time Series values is a quite common problem in practice. Predicting the weather for the next week, the price of Bitcoins tomorrow, the number of your sales during Chrismas and future heart failure are common examples.

Time Series data introduces a "hard dependency" on previous time steps, so the assumption that independence of observations doesn't hold. What are some of the properties that a Time Series can have?

Stationarity, **seasonality**, and **autocorrelation** are some of the properties of the Time Series you might be interested in.

[192] https://colab.research.google.com/drive/1lUwtvOInzoaNC5eBMljRMVk1K9zcKD-b

[193] https://github.com/curiousily/Deep-Learning-For-Hackers

[194] https://en.wikipedia.org/wiki/Time_series

A Times Series is said to be **stationary** when the mean and variance remain constant over time. A Time Series has a **trend** if the mean is varying over time. Often you can eliminate it and make the series stationary by applying log transformation(s).

Seasonality refers to the phenomenon of variations at specific time-frames. eg people buying more Christmas trees during Christmas (who would've thought). A common approach to eliminating seasonality is to use differencing[195].

Autocorrelation[196] refers to the correlation between the current value with a copy from a previous time (lag).

Why we would want to seasonality, trend and have a stationary Time Series? This is required data preprocessing step for Time Series forecasting with classical methods like ARIMA models[197]. Luckily, we'll do our modeling using Recurrent Neural Networks.

Recurrent Neural Networks

Recurrent neural networks (RNNs) can predict the next value(s) in a sequence or classify it. A sequence is stored as a matrix, where each row is a feature vector that describes it. Naturally, the order of the rows in the matrix is important.

RNNs are a really good fit for solving Natural Language Processing (NLP) tasks where the words in a text form sequences and their position matters. That said, cutting edge NLP uses the Transformer[198] for most (if not all) tasks.

As you might've already guessed, Time Series is just one type of a sequence. We'll have to cut the Time Series into smaller sequences, so our RNN models can use them for training. But how do we train RNNs?

First, let's develop an intuitive understanding of what recurrent means. RNNs contain loops. Each unit has a state and receives two inputs - states from the previous layer and the stats from this layer from the previous time step.

The Backpropagation algorithm[199] breaks down when applied to RNNs because of the recurrent connections. Unrolling the network, where copies of the neurons that have recurrent connections are created, can solve this problem. This converts the RNN into a regular Feedforward Neural Net, and classic Backpropagation can be applied. The modification is known as Backpropagation through time[200].

[195] https://www.quora.com/What-is-the-purpose-of-differencing-in-time-series-models
[196] https://en.wikipedia.org/wiki/Autocorrelation
[197] https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average
[198] https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)
[199] https://en.wikipedia.org/wiki/Backpropagation
[200] https://en.wikipedia.org/wiki/Backpropagation_through_time

Problems with Classical RNNs

Unrolled Neural Networks can get very deep (that's what he said), which creates problems for the gradient calculations. The weights can become very small (Vanishing gradient problem[201]) or very large (Exploding gradient problem[202]).

Classic RNNs also have a problem with their memory (long-term dependencies), too. The begging of the sequences we use for training tends to be "forgotten" because of the overwhelming effect of more recent states.

In practice, those problems are solved by using gated RNNs. They can store information for later use, much like having a memory. Reading, writing, and deleting from the memory are learned from the data. The two most commonly used gated RNNs are Long Short-Term Memory Networks[203] and Gated Recurrent Unit Neural Networks[204].

Time Series Prediction with LSTMs

We'll start with a simple example of forecasting the values of the Sine function[205] using a simple LSTM network.

Setup

Let's start with the library imports and setting seeds:

```
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

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

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

```

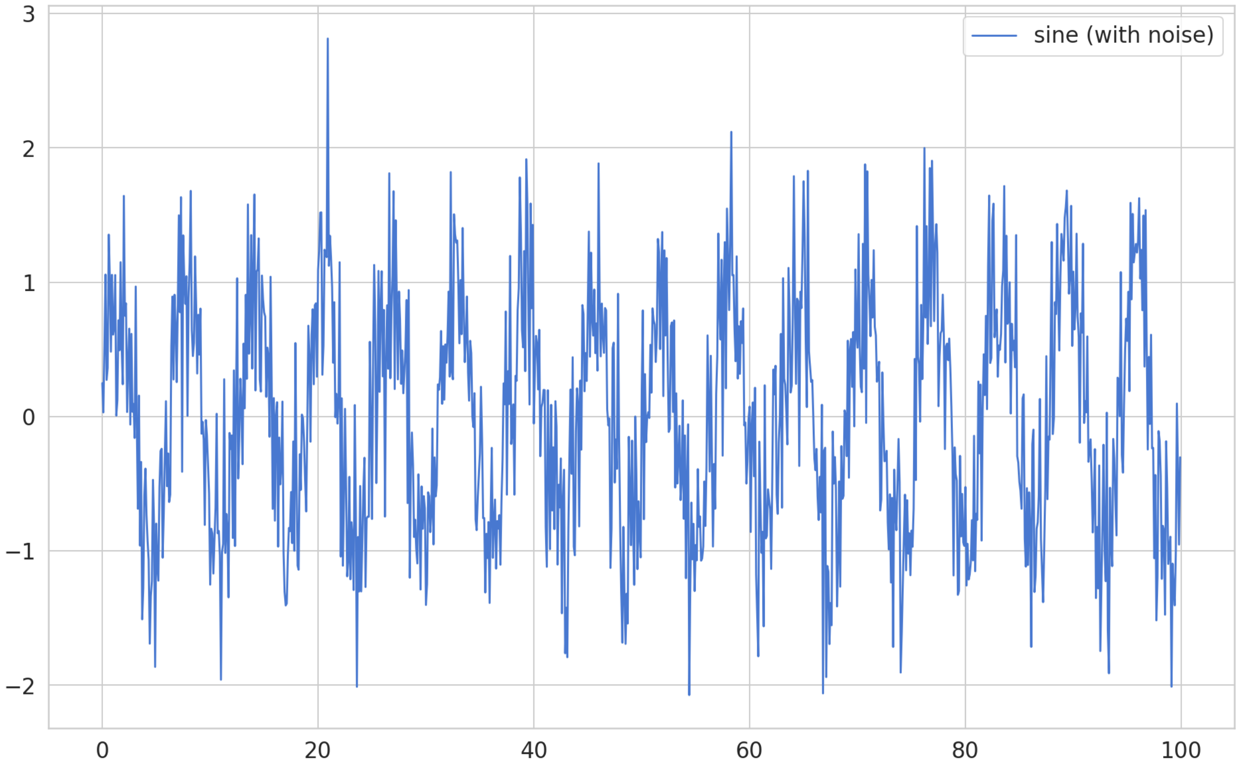
[201] https://en.wikipedia.org/wiki/Vanishing_gradient_problem
[202] (https://en.wikipedia.org/wiki/Vanishing_gradient_problem)
[203] https://en.wikipedia.org/wiki/Long_short-term_memory
[204] https://en.wikipedia.org/wiki/Gated_recurrent_unit
[205] https://en.wikipedia.org/wiki/Sine

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

RANDOM_SEED = 42

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

Data

We'll generate *1,000* values from the sine function and use that as training data. But, we'll add a little bit of *zing* to it:

```
time = np.arange(0, 100, 0.1)
sin = np.sin(time) + np.random.normal(scale=0.5, size=len(time))
```



A random value, drawn from a normal distribution, is added to each data point. That'll make the job of our model a bit harder.

Data Preprocessing

We need to "chop the data" into smaller sequences for our model. But first, we'll split it into training and test data:

```
df = pd.DataFrame(dict(sine=sin), index=time, columns=['sine'])

train_size = int(len(df) * 0.8)
test_size = len(df) - train_size
train, test = df.iloc[0:train_size], df.iloc[train_size:len(df)]
print(len(train), len(test))
```

```
800 200
```

Preparing the data for Time Series forecasting (LSTMs in particular) can be tricky. Intuitively, we need to predict the value at the current time step by using the history (*n* time steps from it). Here's a generic function that does the job:

```
def create_dataset(X, y, time_steps=1):
    Xs, ys = [], []
    for i in range(len(X) - time_steps):
        v = X.iloc[i:(i + time_steps)].values
        Xs.append(v)
        ys.append(y.iloc[i + time_steps])
    return np.array(Xs), np.array(ys)
```

The beauty of this function is that it works with univariate (single feature) and multivariate (multiple features) Time Series data. Let's use a history of 10 time steps to make our sequences:

```
time_steps = 10

## reshape to [samples, time_steps, n_features]

X_train, y_train = create_dataset(train, train.sine, time_steps)
X_test, y_test = create_dataset(test, test.sine, time_steps)

print(X_train.shape, y_train.shape)
```

```
(790, 10, 1) (790,)
```

We have our sequences in the shape `(samples, time_steps, features)`. How can we use them to make predictions?

Modeling

Training an LSTM model in Keras is easy. We'll use the LSTM layer[206] in a sequential model to make our predictions:

[206] https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM

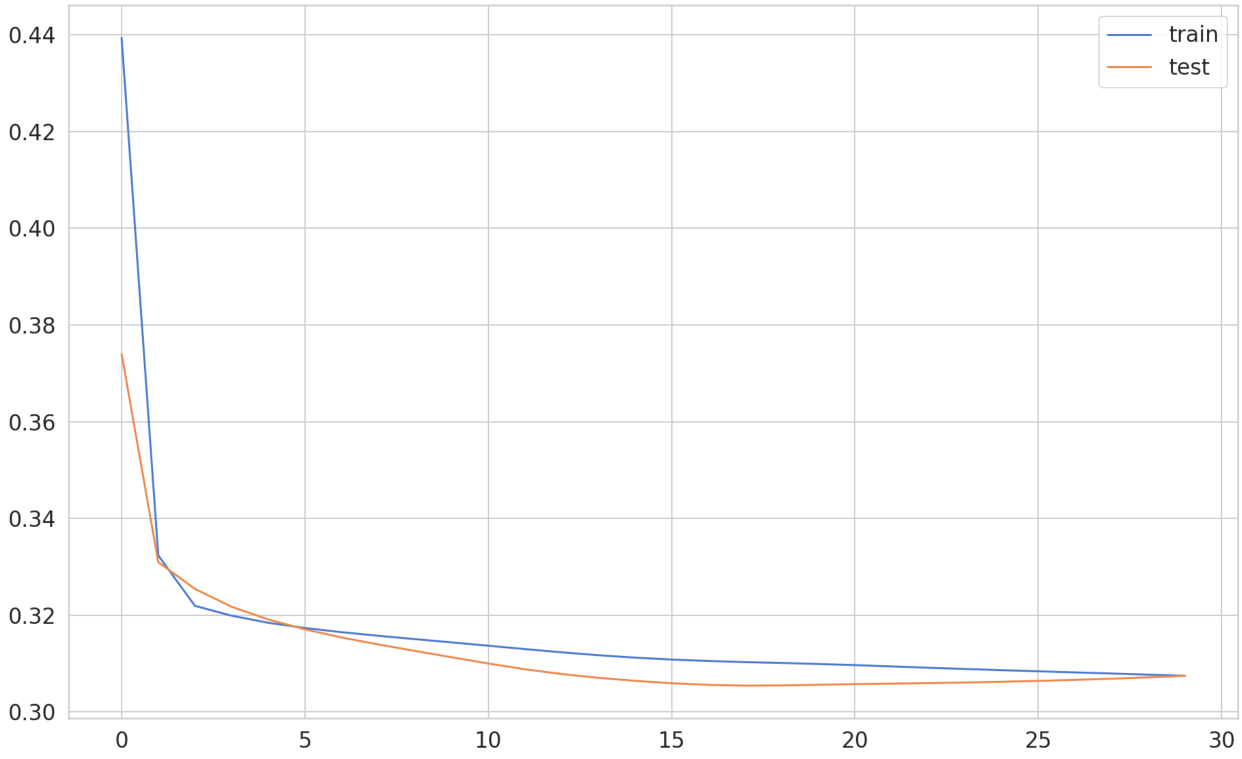
```
model = keras.Sequential()
model.add(keras.layers.LSTM(
  units=128,
  input_shape=(X_train.shape[1], X_train.shape[2])
))
model.add(keras.layers.Dense(units=1))
model.compile(
  loss='mean_squared_error',
  optimizer=keras.optimizers.Adam(0.001)
)
```

The LSTM layer expects the number of time steps and the number of features to work properly. The rest of the model looks like a regular regression model. How do we train a LSTM model?

Training

The most important thing to remember when training Time Series models is to not shuffle the data (the order of the data matters). The rest is pretty standard:

```
history = model.fit(
    X_train, y_train,
    epochs=30,
    batch_size=16,
    validation_split=0.1,
    verbose=1,
    shuffle=False
)
```



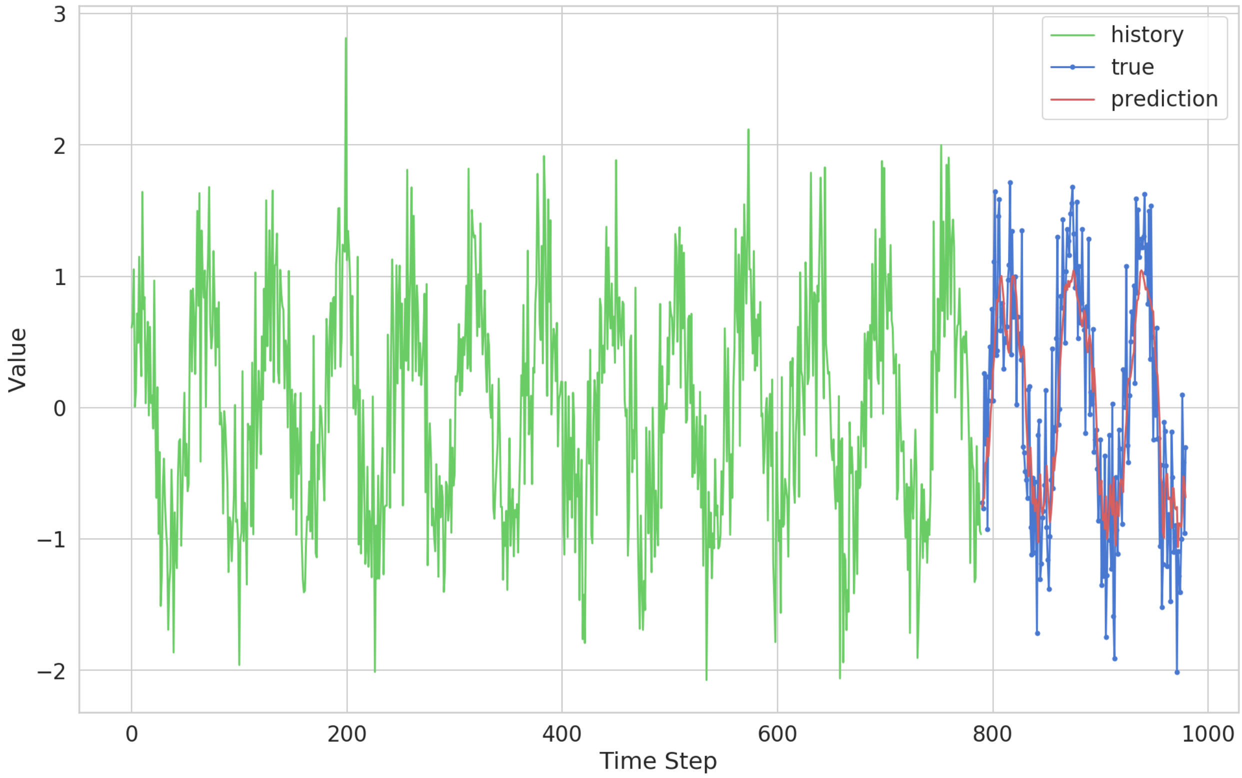
Our dataset is pretty simple and contains the randomness from our sampling. After about 15 epochs, the model is pretty much-done learning.

Evaluation

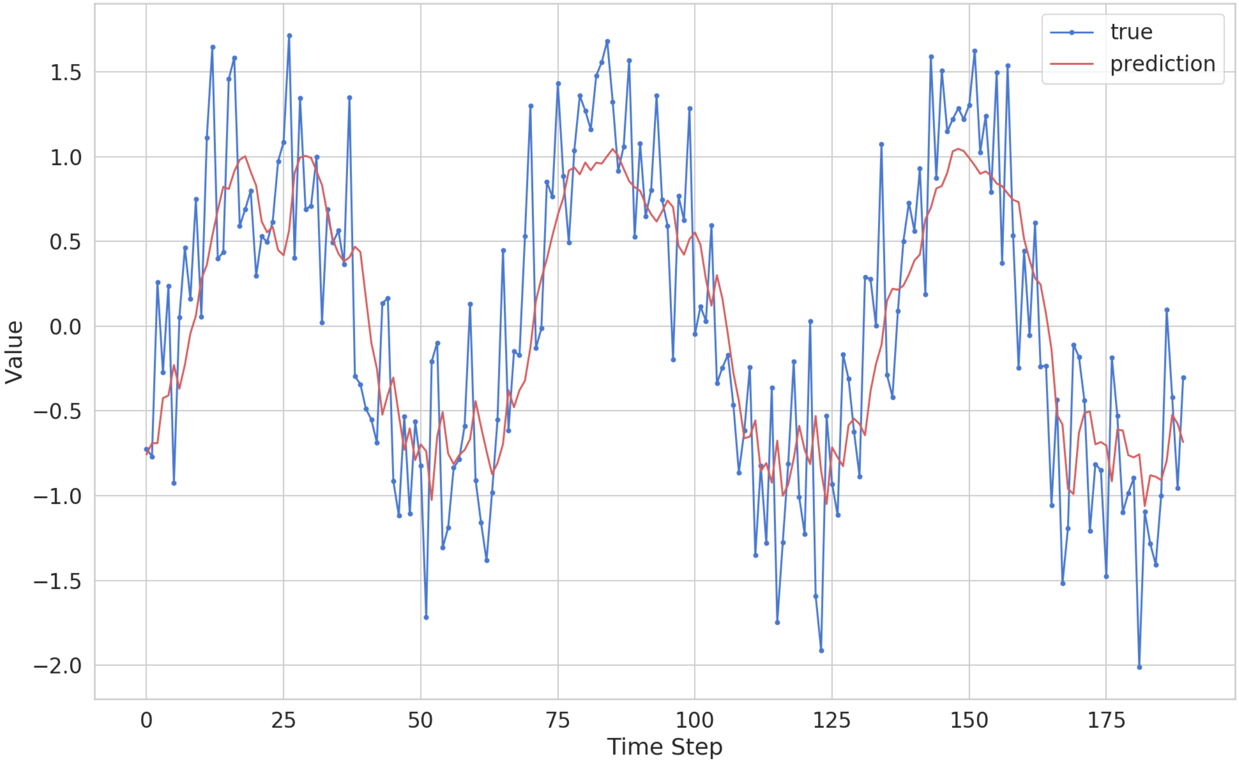
Let’s take some predictions from our model:

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

We can plot the predictions over the true values from the Time Series:



Our predictions look really good on this scale. Let’s zoom in:



The model seems to be doing a great job of capturing the general pattern of the data. It fails to capture random fluctuations, which is a good thing (it generalizes well).

Conclusion

Congratulations! You made your first Recurrent Neural Network model! You also learned how to preprocess Time Series data, something that trips a lot of people.

- Time Series
- Recurrent Neural Networks
- Time Series Prediction with LSTMs

We've just scratched the surface of Time Series data and how to use Recurrent Neural Networks. Some interesting applications are Time Series forecasting, (sequence) classification and anomaly detection. The fun part is just getting started!

Run the complete notebook in your browser[207]

The complete project on GitHub[208]

[207] https://colab.research.google.com/drive/1lUwtvOInzoaNC5eBMljRMVk1K9zcKD-b
[208] https://github.com/curiousily/Deep-Learning-For-Hackers

References

- TensorFlow - Time series forecasting[209]
- Understanding LSTM Networks[210]

[209] https://www.tensorflow.org/tutorials/structured_data/time_series
[210] https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Cryptocurrency price prediction using LSTMs

> TL;DR Build and train an Bidirectional LSTM Deep Neural Network for Time Series prediction in TensorFlow 2. Use the model to predict the future Bitcoin price.

Complete source code in Google Colaboratory Notebook[211]

This time you'll build a basic Deep Neural Network model to predict Bitcoin price based on historical data. **You can use the model however you want, but you carry the risk for your actions**.

You might be asking yourself something along the lines:

> Can I still get rich with cryptocurrency?

Of course, the answer is fairly nuanced. Here, we'll have a look at how you might build a model to help you along the crazy journey.

Or you might be having money problems? Here is one possible solution[212]:

Here is the plan:

1. Cryptocurrency data overview
2. Time Series
3. Data preprocessing
4. Build and train LSTM model in TensorFlow 2
5. Use the model to predict future Bitcoin price

Data Overview

Our dataset comes from Yahoo! Finance[213] and covers all available (at the time of this writing) data on Bitcoin-USD price. Let's load it into a Pandas dataframe:

[211] https://colab.research.google.com/drive/1wWvtA5RC6-is6J8W86wzK52Knr3N1Xbm

[212] https://www.youtube.com/watch?v=C-m3RtoguAQ

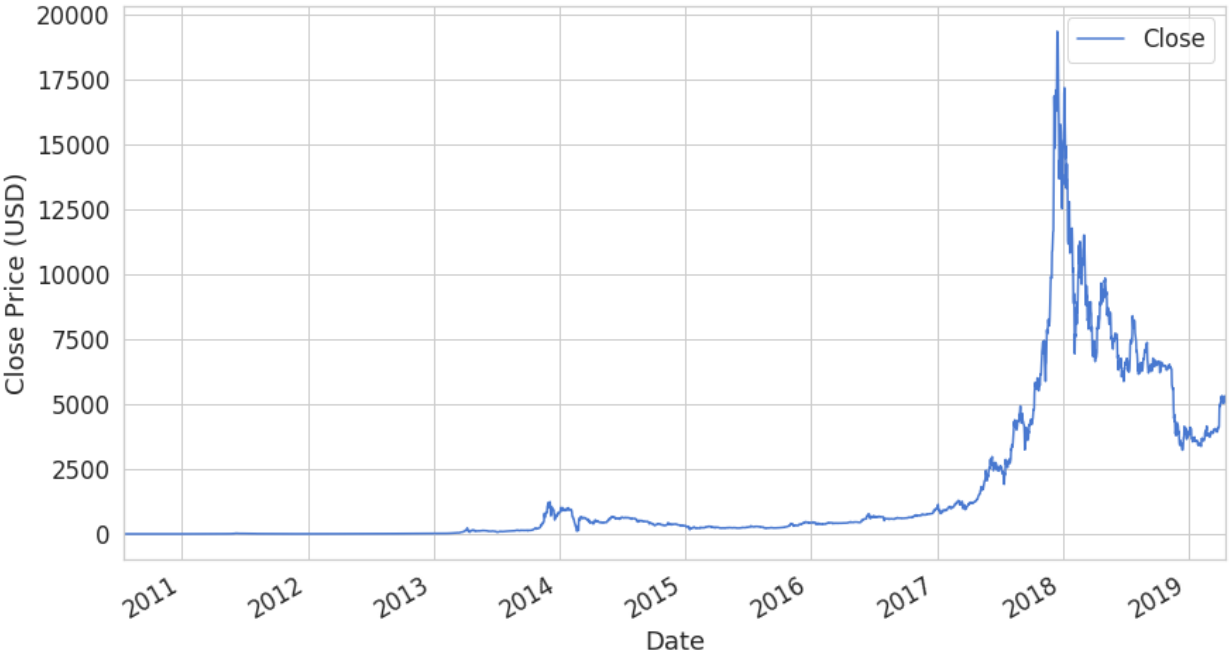
[213] https://finance.yahoo.com/quote/BTC-USD/history?period1=1279314000&period2=1556053200&interval=1d&filter=history&frequency=1d

```
csv_path = "https://raw.githubusercontent.com/curiousily/Deep-Learning-For-Hackers/m\
aster/data/3.stock-prediction/BTC-USD.csv"
df = pd.read_csv(csv_path, parse_dates=['Date'])
df = df.sort_values('Date')
```

Note that we sort the data by Date just in case. Here is a sample of the data we're interested in:

Date	Close
2010-07-16	0.04951
2010-07-17	0.08584
2010-07-18	0.08080
2010-07-19	0.07474
2010-07-20	0.07921

We have a total of *3201* data points representing Bitcoin-USD price for *3201* days (~9 years). We're interested in predicting the closing price for future dates.



Of course, Bitcoin made some people really rich[214] and for some went really poor. The question remains though, will it happen again? Let's have a look at what one possible model thinks about that. Shall we?

[214] https://www.reddit.com/r/Bitcoin/comments/7j653t/what_does_it_feel_to_be_rich_beacuse_of_bitcoin/

Time Series

Our dataset is somewhat different from our previous examples. The data is sorted by time and recorded at equal intervals (1 day). Such a sequence of data is called *Time Series*[215].

Temporal datasets are quite common in practice. Your energy consumption and expenditure (calories in, calories out), weather changes, stock market, analytics gathered from the users for your product/app and even your (possibly in love) heart produce *Time Series*.

You might be interested in a plethora of properties regarding your Time Series - **stationarity**, **seasonality** and **autocorrelation** are some of the most well known.

Autocorrelation is the correlation of data points separated by some interval (known as lag).

Seasonality refers to the presence of some cyclical pattern at some interval (no, it doesn't have to be every spring).

A time series is said to be **stationarity** if it has constant mean and variance. Also, the covariance is independent of the time.

One obvious question you might ask yourself while watching at Time Series data is: "Does the value of the current time step affects the next one?" a.k.a. *Time Series forecasting*.

There are many approaches that you can use for this purpose. But we'll build a Deep Neural Network that does some forecasting for us and use it to predict future Bitcoin price.

Modeling

All models we've built so far do not allow for operating on sequence data. Fortunately, we can use a special class of Neural Network models known as Recurrent Neural Networks (RNNs)[216] just for this purpose. *RNNs* allow using the output from the model as a new input for the same model. The process can be repeated indefinitely.

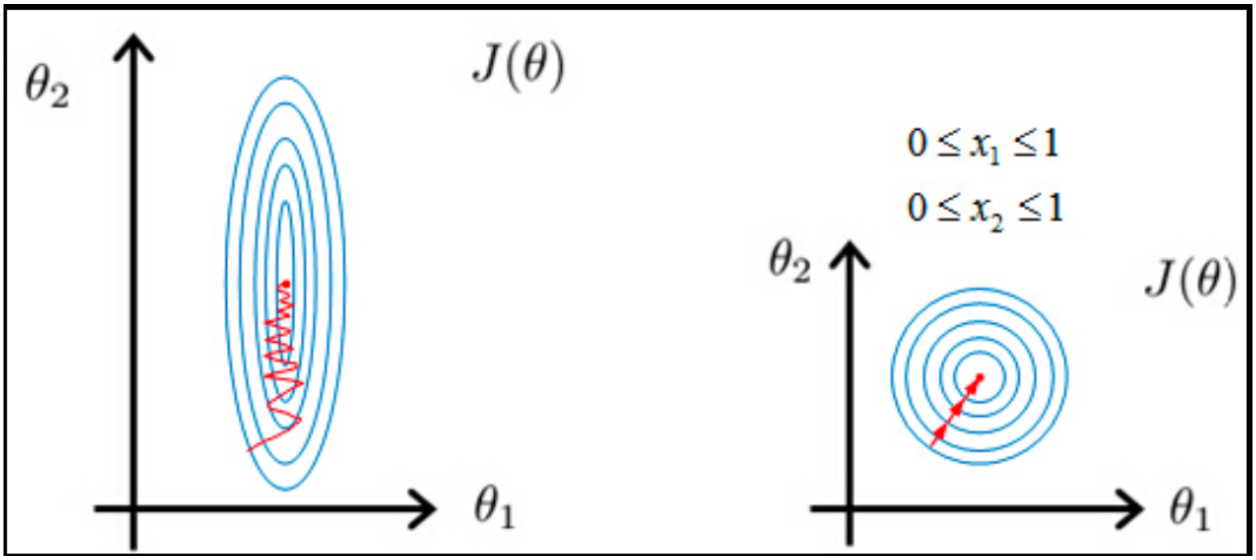
One serious limitation of *RNNs* is the inability of capturing long-term dependencies[217] in a sequence (e.g. Is there a dependency between today's price and that 2 weeks ago?). One way to handle the situation is by using an **Long short-term memory (LSTM)** variant of *RNN*.

The default LSTM[218] behavior is remembering information for prolonged periods of time. Let's see how you can use LSTM in Keras.

Data preprocessing

First, we're going to squish our price data in the range [0, 1]. Recall that this will help our optimization algorithm converge faster:

[215] https://en.wikipedia.org/wiki/Time_series
[216] https://en.wikipedia.org/wiki/Recurrent_neural_network
[217] https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[218] https://en.wikipedia.org/wiki/Long_short-term_memory



source: Andrew Ng[219]

We're going to use the MinMaxScaler[220] from scikit learn[221]:

```
scaler = MinMaxScaler()

close_price = df.Close.values.reshape(-1, 1)

scaled_close = scaler.fit_transform(close_price)
```

The scaler expects the data to be shaped as (x, y), so we add a dummy dimension using reshape[222] before applying it.

Let's also remove NaNs since our model won't be able to handle them well:

```
scaled_close = scaled_close[~np.isnan(scaled_close)]
scaled_close = scaled_close.reshape(-1, 1)
```

We use isnan[223] as a mask to filter out NaN values. Again we reshape the data after removing the NaNs.

Making sequences

LSTMs expect the data to be in 3 dimensions. We need to split the data into sequences of some preset length. The shape we want to obtain is:

[219] https://www.andrewng.org/
[220] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
[221] https://scikit-learn.org/stable/index.html
[222] https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html
[223] https://docs.scipy.org/doc/numpy/reference/generated/numpy.isnan.html

```
[batch_size, sequence_length, n_features]
```

We also want to save some data for testing. Let's build some sequences:

```
SEQ_LEN = 100

def to_sequences(data, seq_len):
    d = []

    for index in range(len(data) - seq_len):
        d.append(data[index: index + seq_len])

    return np.array(d)

def preprocess(data_raw, seq_len, train_split):

    data = to_sequences(data_raw, seq_len)

    num_train = int(train_split * data.shape[0])

    X_train = data[:num_train, :-1, :]
    y_train = data[:num_train, -1, :]

    X_test = data[num_train:, :-1, :]
    y_test = data[num_train:, -1, :]

    return X_train, y_train, X_test, y_test


X_train, y_train, X_test, y_test =\
 preprocess(scaled_close, SEQ_LEN, train_split = 0.95)
```

The process of building sequences works by creating a sequence of a specified length at position 0. Then we shift one position to the right (e.g. 1) and create another sequence. The process is repeated until all possible positions are used.

We save 5% of the data for testing. The datasets look like this:

```
X_train.shape
```

```
(2945, 99, 1)
```

```
X_test.shape
```

```
(156, 99, 1)
```

Our model will use 2945 sequences representing 99 days of Bitcoin price changes each for training. We're going to predict the price for 156 days in the future (from our model POV).

Building LSTM model

We're creating a 3 layer LSTM[224] Recurrent Neural Network. We use Dropout[225] with a rate of 20% to combat overfitting during training:

```
DROPOUT = 0.2
WINDOW_SIZE = SEQ_LEN - 1

model = keras.Sequential()

model.add(Bidirectional(
  CuDNNLSTM(WINDOW_SIZE, return_sequences=True),
  input_shape=(WINDOW_SIZE, X_train.shape[-1])
))
model.add(Dropout(rate=DROPOUT))

model.add(Bidirectional(
  CuDNNLSTM((WINDOW_SIZE * 2), return_sequences=True)
))
model.add(Dropout(rate=DROPOUT))

model.add(Bidirectional(
  CuDNNLSTM(WINDOW_SIZE, return_sequences=False)
))

model.add(Dense(units=1))

model.add(Activation('linear'))
```

You might be wondering about what the deal with Bidirectional[226] and CuDNNLSTM is?

Bidirectional RNNs[227] allows you to train on the sequence data in forward and backward (reversed) direction. In practice, this approach works well with LSTMs.

[224] https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/LSTM
[225] https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Dropout
[226] https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Bidirectional
[227] https://maxwell.ict.griffith.edu.au/spl/publications/papers/ieeesp97_schuster.pdf

CuDNNLSTM[228] is a “Fast LSTM implementation backed by cuDNN”. Personally, I think it is a good example of leaky abstraction, but it is crazy fast!

Our output layer has a single neuron (predicted Bitcoin price). We use Linear activation function[229] which activation is proportional to the input.

Training

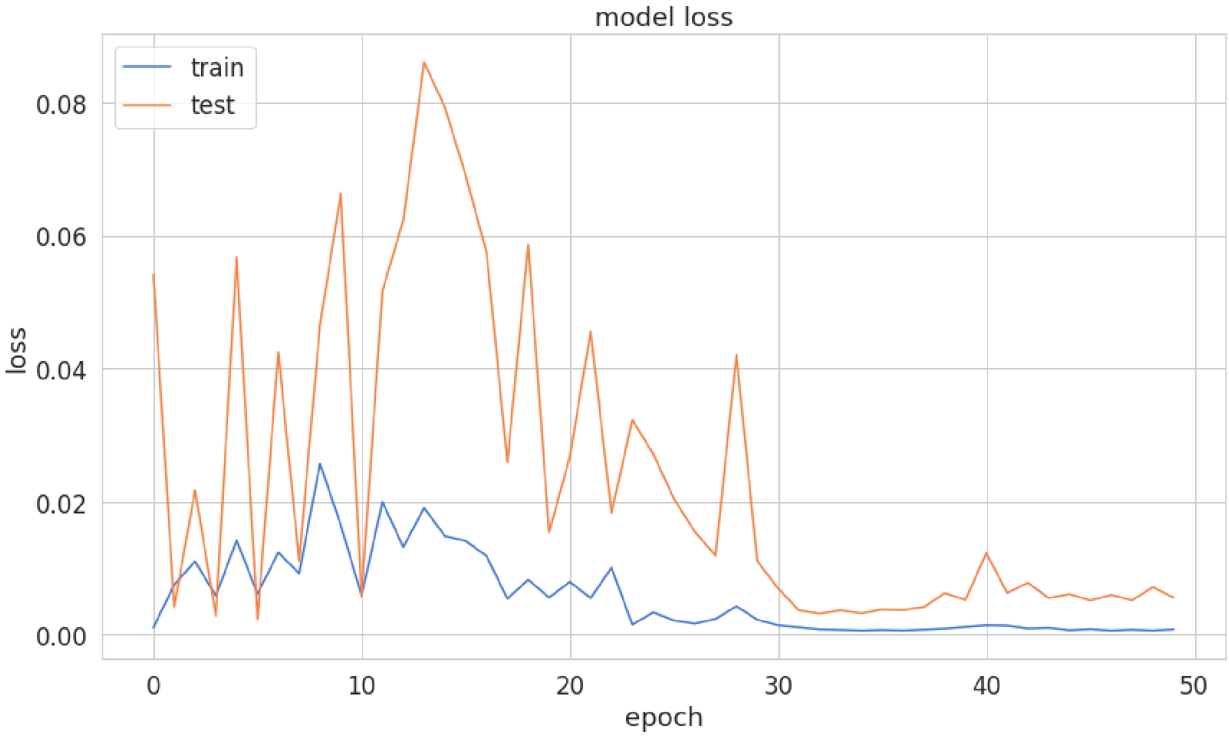
We’ll use Mean Squared Error[230] as a loss function and Adam[231] optimizer.

```
BATCH_SIZE = 64

model.compile(
    loss='mean_squared_error',
    optimizer='adam'
)

history = model.fit(
    X_train,
    y_train,
    epochs=50,
    batch_size=BATCH_SIZE,
    shuffle=False,
    validation_split=0.1
)
```

Note that we do not want to shuffle the training data since we’re using Time Series.

After a lightning-fast training (thanks Google for the free T4 GPUs), we have the following training loss:

[228] https://www.tensorflow.org/api_docs/python/tf/keras/layers/CuDNNLSTM
[229] https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#linear
[230] https://en.wikipedia.org/wiki/Mean_squared_error
[231] https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/optimizers/Adam



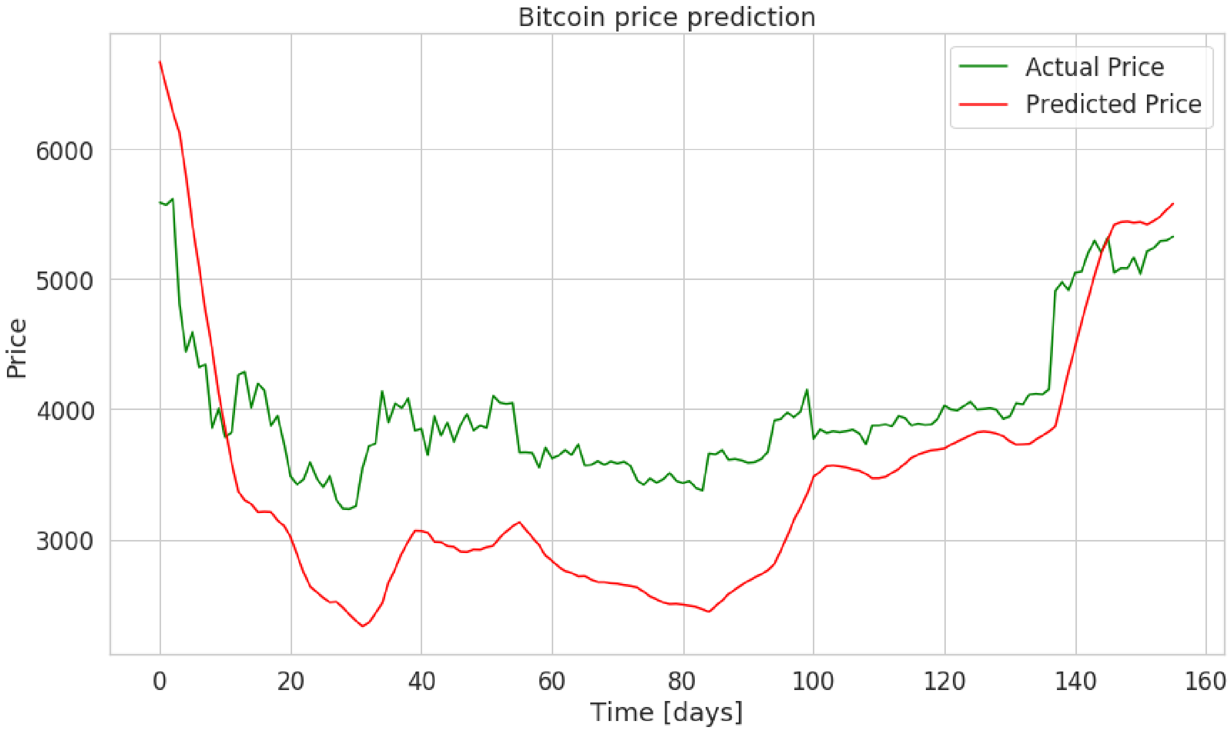
Predicting Bitcoin price

Let’s make our model predict Bitcoin prices!

```
y_hat = model.predict(X_test)
```

We can use our scaler to invert the transformation we did so the prices are no longer scaled in the [0, 1] range.

```
y_test_inverse = scaler.inverse_transform(y_test)
y_hat_inverse = scaler.inverse_transform(y_hat)
```



Our rather succinct model seems to do well on the test data. Care to try it on other currencies?

Conclusion

Congratulations, you just built a Bidirectional LSTM Recurrent Neural Network in TensorFlow 2. Our model (and preprocessing “pipeline”) is pretty generic and can be used for other datasets.

Complete source code in Google Colaboratory Notebook[232]

One interesting direction of future investigation might be analyzing the correlation between different cryptocurrencies and how would that affect the performance of our model.

[232] https://colab.research.google.com/drive/1wWvtA5RC6-is6J8W86wzK52Knr3N1Xbm

Demand Prediction for Multivariate Time Series with LSTMs

> TL;DR Learn how to predict demand using Multivariate Time Series Data. Build a Bidirectional LSTM Neural Network in Keras and TensorFlow 2 and use it to make predictions.

One of the most common applications of Time Series models is to predict future values. How the stock market is going to change? How much will 1 Bitcoin cost tomorrow? How much coffee are you going to sell next month?

This guide will show you how to use Multivariate (many features) Time Series data to predict future demand. You'll learn how to preprocess and scale the data. And you're going to build a Bidirectional LSTM Neural Network to make the predictions.

Here are the steps you'll take:

- Data
- Feature Engineering
- Exploration
- Preprocessing
- Predicting Demand
- Evaluation

Run the complete notebook in your browser[233]

The complete project on GitHub[234]

Data

Our data London bike sharing dataset[235] is hosted on Kaggle. It is provided by Hristo Mavrodiev[236]. Thanks!

> A bicycle-sharing system, public bicycle scheme, or public bike share (PBS) scheme, is a service in which bicycles are made available for shared use to individuals on a short term basis for a price or free. - Wikipedia[237]

[233] https://colab.research.google.com/drive/1k3PLdczAJOIrIprfhjZ-IRXzNhFJ_OTN
[234] https://github.com/curiousily/Deep-Learning-For-Hackers
[235] https://www.kaggle.com/hmavrodiev/london-bike-sharing-dataset
[236] https://www.kaggle.com/hmavrodiev
[237] https://en.wikipedia.org/wiki/Bicycle-sharing_system

Our goal is to predict the number of future bike shares given the historical data of London bike shares. Let's download the data:

```
!gdown --id 1nPw071R3tZi4zqVcmXA6kXVTe43Ex6K3 --output london_bike_sharing.csv
```

and load it into a Pandas data frame:

```
df = pd.read_csv(
  "london_bike_sharing.csv",
  parse_dates=['timestamp'],
  index_col="timestamp"
)
```

Pandas is smart enough to parse the timestamp strings as DateTime objects. What do we have? We have 2 years of bike-sharing data, recorded at regular intervals (1 hour). And in terms of the number of rows:

```
df.shape
```

```
(17414, 9)
```

That might do. What features do we have?

- timestamp - timestamp field for grouping the data
- cnt - the count of a new bike shares
- t1 - real temperature in C
- t2 - temperature in C "feels like"
- hum - humidity in percentage
- wind_speed - wind speed in km/h
- weather_code - category of the weather
- is_holiday - boolean field - 1 holiday / 0 non holiday
- is_weekend - boolean field - 1 if the day is weekend
- season - category field meteorological seasons: 0-spring ; 1-summer; 2-fall; 3-winter.

How well can we predict future demand based on the data?

Feature Engineering

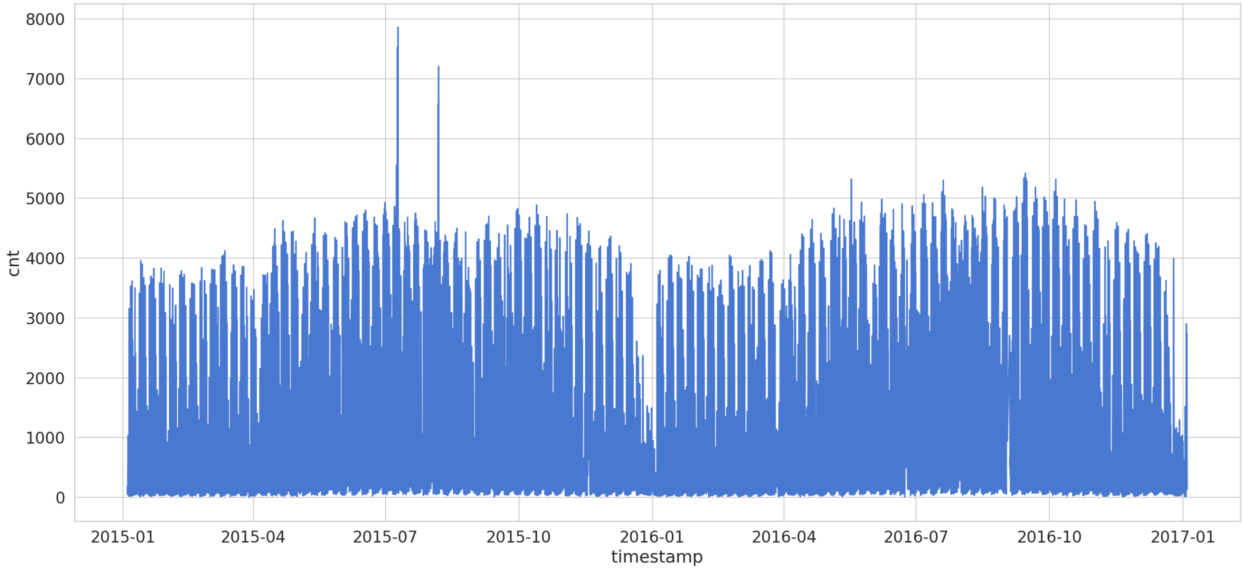
We'll do a little bit of engineering:

```
df['hour'] = df.index.hour
df['day_of_month'] = df.index.day
df['day_of_week'] = df.index.dayofweek
df['month'] = df.index.month
```

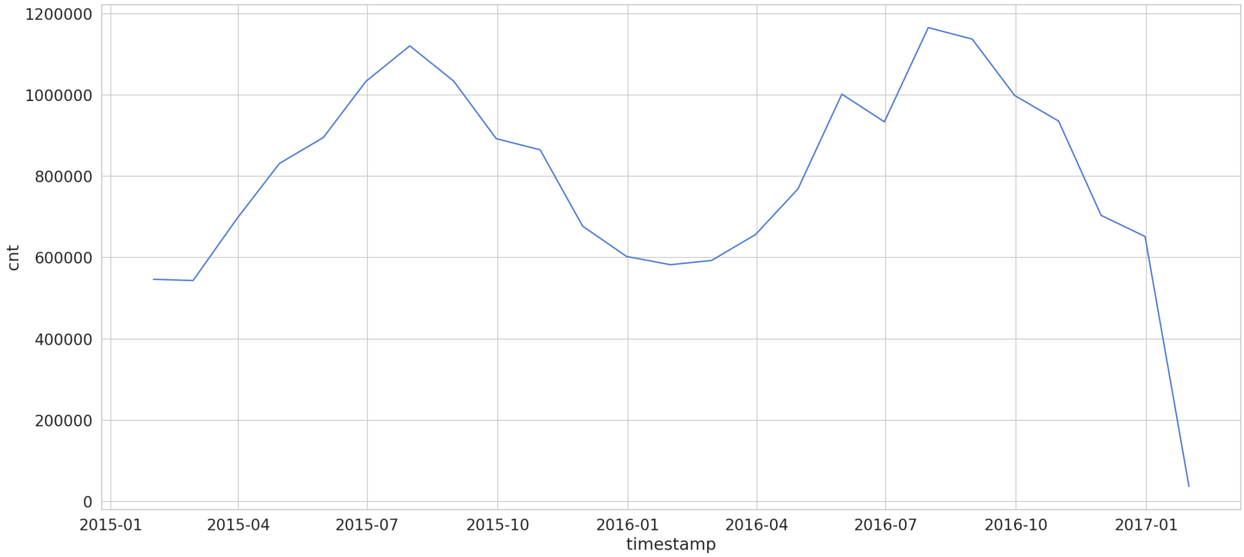
All new features are based on the timestamp. Let's dive deeper into the data.

Exploration

Let's start simple. Let's have a look at the bike shares over time:

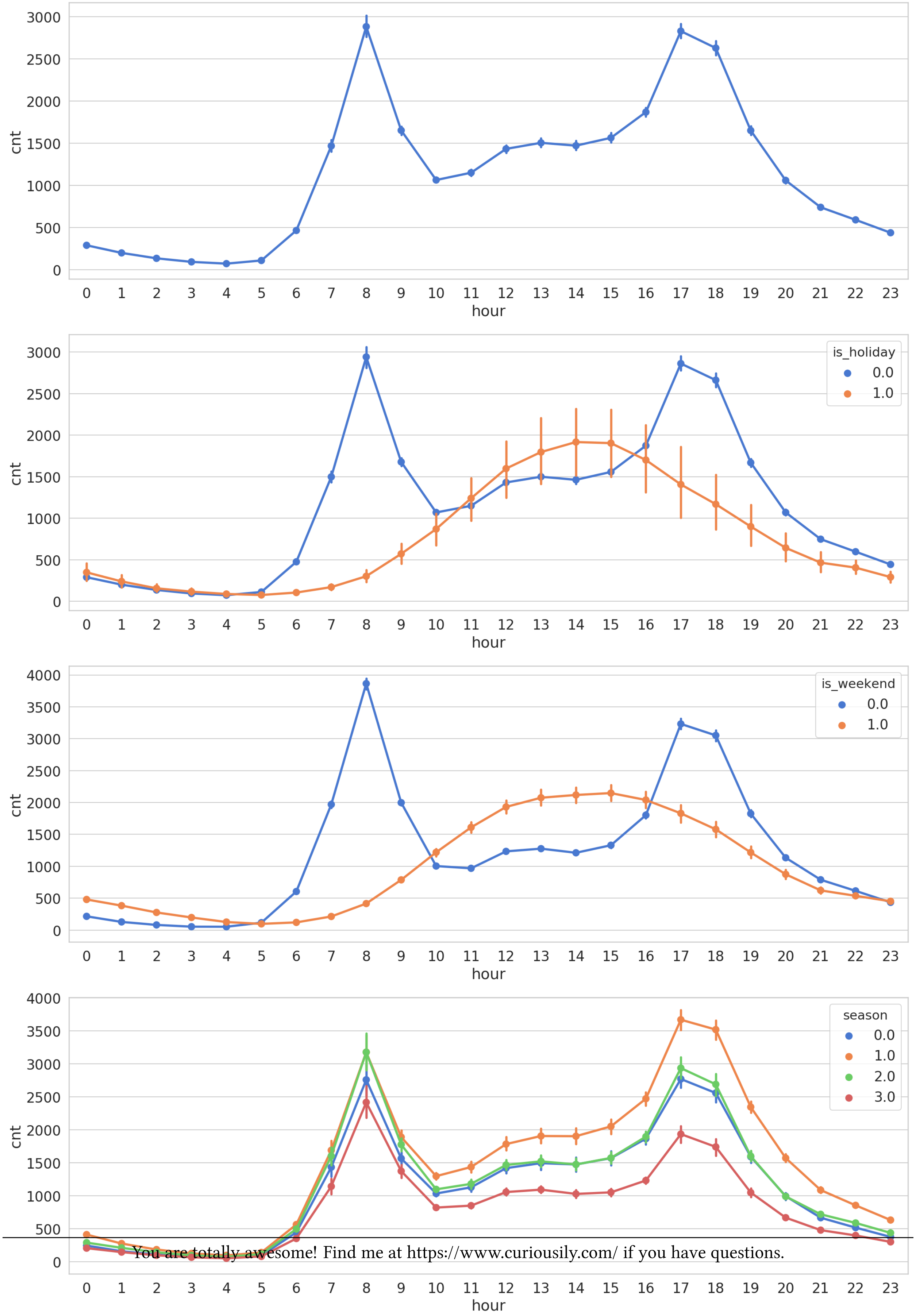


That's a bit too crowded. Let's have a look at the same data on a monthly basis:

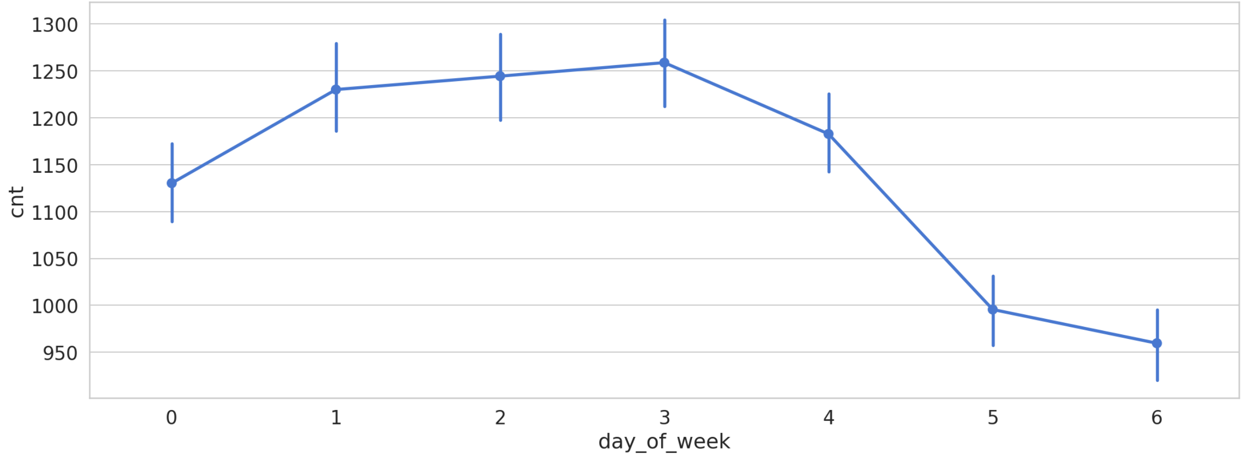


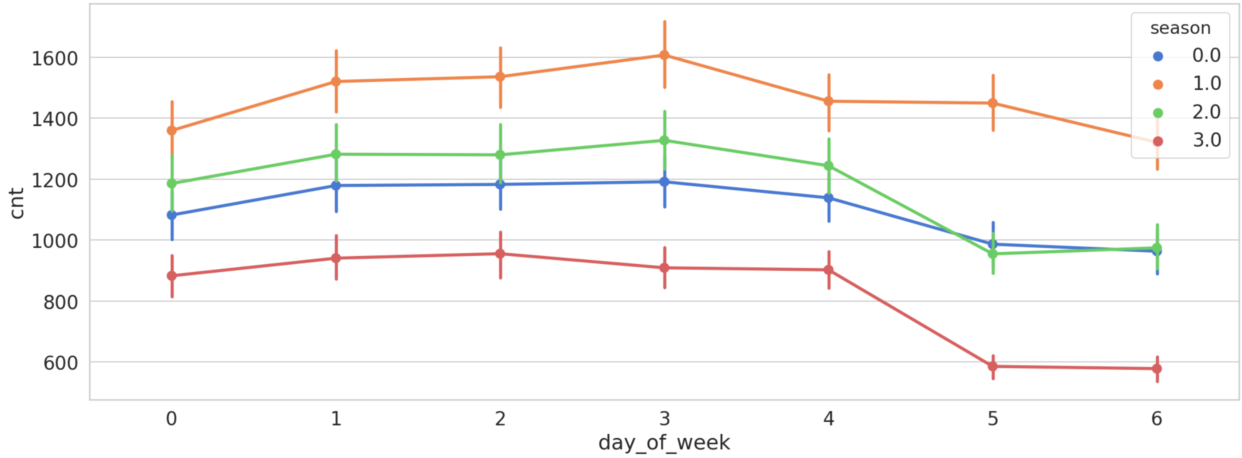
Our data seems to have a strong seasonality component. Summer months are good for business.

How about the bike shares by the hour:


cnt
0
500
1000
1500
2000
2500
3000
hour
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
is_holiday
0.0
1.0
is_weekend
0.0
1.0
3500
4000
season
0.0
1.0
2.0
3.0

The hours with most bike shares differ significantly based on a weekend or not days. Workdays contain two large spikes during the morning and late afternoon hours (people pretend to work in between). On weekends early to late afternoon hours seem to be the busiest.





Looking at the data by day of the week shows a much higher count on the number of bike shares.

Our little feature engineering efforts seem to be paying off. The new features separate the data very well.

Preprocessing

We'll use the last 10% of the data for testing:

```
train_size = int(len(df) * 0.9)
test_size = len(df) - train_size
train, test = df.iloc[0:train_size], df.iloc[train_size:len(df)]
print(len(train), len(test))
```

```
15672 1742
```

We'll scale some of the features we're using for our modeling:

```
f_columns = ['t1', 't2', 'hum', 'wind_speed']
```

```
f_transformer = RobustScaler()

f_transformer = f_transformer.fit(train[f_columns].to_numpy())

train.loc[:, f_columns] = f_transformer.transform(
  train[f_columns].to_numpy()
)

test.loc[:, f_columns] = f_transformer.transform(
  test[f_columns].to_numpy()
)
```

We'll also scale the number of bike shares too:

```
cnt_transformer = RobustScaler()

cnt_transformer = cnt_transformer.fit(train[['cnt']])

train['cnt'] = cnt_transformer.transform(train[['cnt']])

test['cnt'] = cnt_transformer.transform(test[['cnt']])
```

To prepare the sequences, we're going to reuse the same `create_dataset()` function:

```
def create_dataset(X, y, time_steps=1):
    Xs, ys = [], []
    for i in range(len(X) - time_steps):
        v = X.iloc[i:(i + time_steps)].values
        Xs.append(v)
        ys.append(y.iloc[i + time_steps])
    return np.array(Xs), np.array(ys)
```

Each sequence is going to contain 10 data points from the history:

```
time_steps = 10

## reshape to [samples, time_steps, n_features]

X_train, y_train = create_dataset(train, train.cnt, time_steps)
X_test, y_test = create_dataset(test, test.cnt, time_steps)

print(X_train.shape, y_train.shape)
```

```
(15662, 10, 13) (15662,)
```

Our data is not in the correct format for training an LSTM model. How well can we predict the number of bike shares?

Predicting Demand

Let's start with a simple model and see how it goes. One layer of Bidirectional[238] LSTM with a Dropout layer[239]:

```
model = keras.Sequential()
model.add(
  keras.layers.Bidirectional(
    keras.layers.LSTM(
      units=128,
      input_shape=(X_train.shape[1], X_train.shape[2])
    )
  )
)
model.add(keras.layers.Dropout(rate=0.2))
model.add(keras.layers.Dense(units=1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

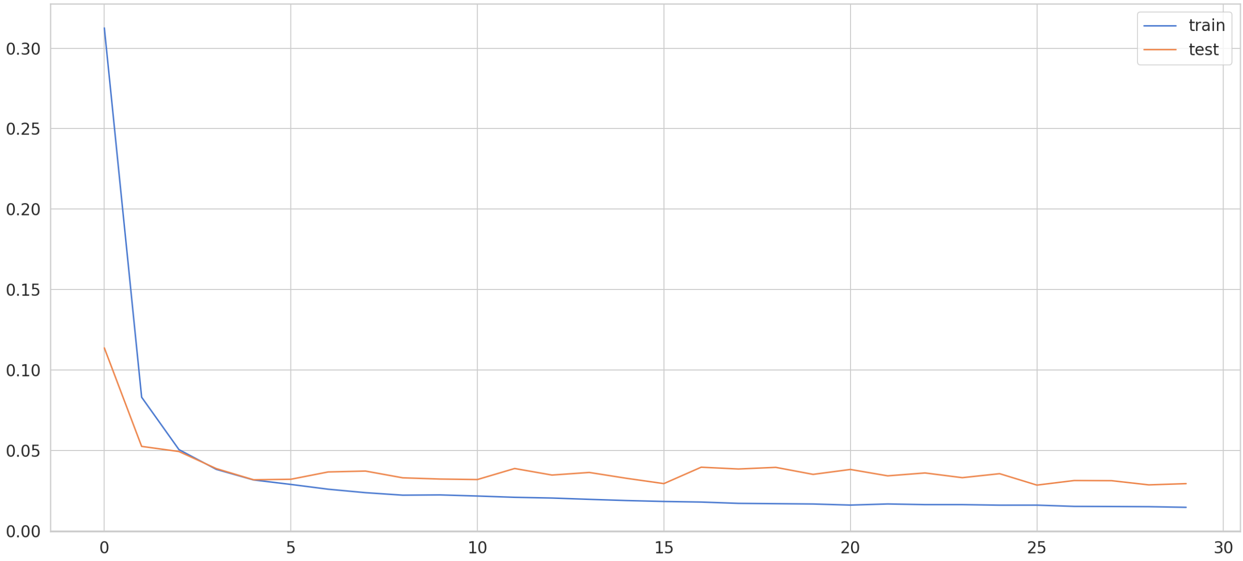
[238] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Bidirectional
[239] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout

Remember to NOT shuffle the data when training:

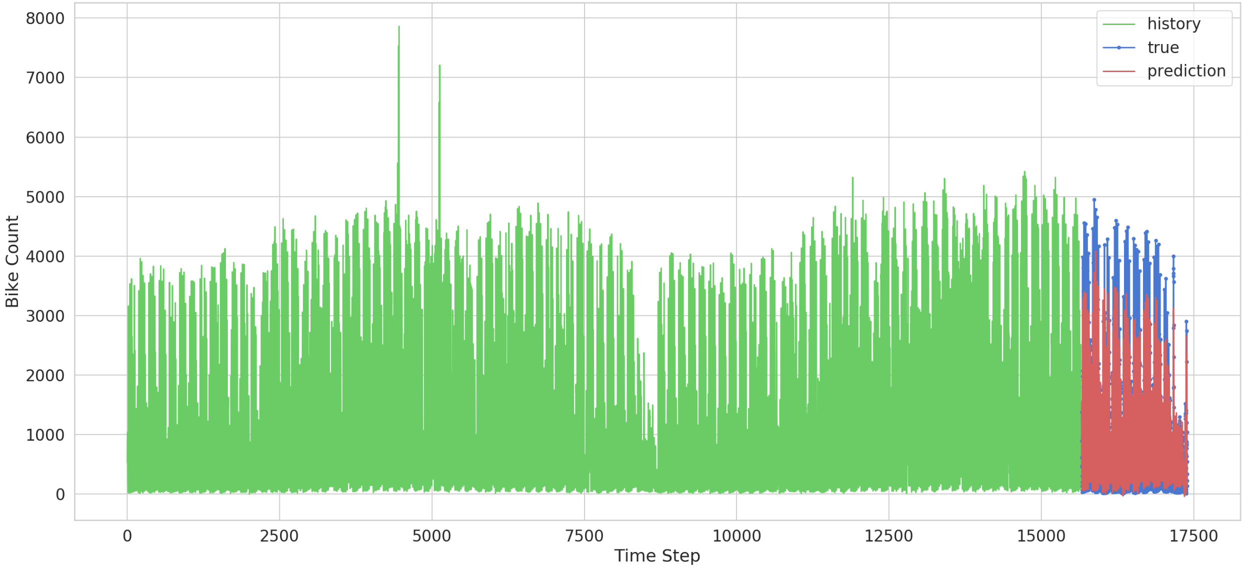
```
history = model.fit(
    X_train, y_train,
    epochs=30,
    batch_size=32,
    validation_split=0.1,
    shuffle=False
)
```

Evaluation

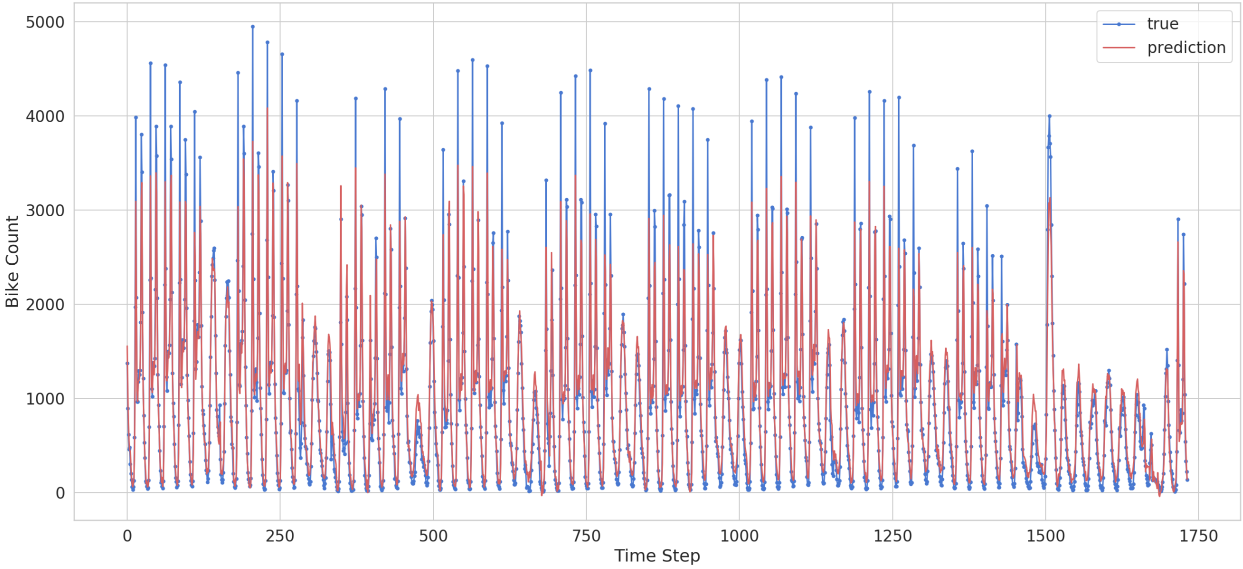
Here's what we have after training our model for *30* epochs:



You can see that the model learns pretty quickly. At about epoch *5*, it is already starting to overfit a bit. You can play around - regularize it, change the number of units, etc. But how well can we predict demand with it?



That might be too much for your eyes. Let's zoom in on the predictions:



Note that our model is predicting only one point in the future. That being said, it is doing very well. Although our model can't really capture the extreme values it does a good job of predicting (understanding) the general pattern.

Conclusion

You just took a real dataset, preprocessed it, and used it to predict bike-sharing demand. You've used a Bidirectional LSTM model to train it on subsequences from the original dataset. You even got some very good results.

Here are the steps you took:

- Data
- Feature Engineering
- Exploration
- Preprocessing
- Predicting Demand
- Evaluation

Run the complete notebook in your browser[240]

The complete project on GitHub[241]

Are there other applications of LSTMs for Time Series data?

References

- TensorFlow - Time series forecasting[242]
- Understanding LSTM Networks[243]
- London bike sharing dataset[244]

[240] https://colab.research.google.com/drive/1k3PLdczAJOIrIprfhjZ-IRXzNhFJ_OTN
[241] https://github.com/curiousily/Deep-Learning-For-Hackers
[242] https://www.tensorflow.org/tutorials/structured_data/time_series
[243] https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[244] https://www.kaggle.com/hmavrodiev/london-bike-sharing-dataset

Time Series Classification for Human Activity Recognition with LSTMs in Keras

> TL;DR Learn how to classify Time Series data from accelerometer sensors using LSTMs in Keras

Can you use Time Series data to recognize user activity from accelerometer data? Your phone/wristband/watch is already doing it. How well can you do it?

We'll use accelerometer data, collected from multiple users, to build a Bidirectional LSTM model and try to classify the user activity. You can deploy/reuse the trained model on any device that has an accelerometer (which is pretty much every smart device).

This is the plan:

- Load Human Activity Recognition Data
- Build LSTM Model for Classification
- Evaluate the Model

Run the complete notebook in your browser[245]

The complete project on GitHub[246]

Human Activity Data

Our data is collected through controlled laboratory conditions. It is provided by the WISDM: WIreless Sensor Data Mining[247] lab.

The data is used in the paper: Activity Recognition using Cell Phone Accelerometers[248]. Take a look at the paper to get a feel of how well some baseline models are performing.

Loading the Data

Let's download the data:

[245] https://colab.research.google.com/drive/1hxq4-A4SZYfKqmqfwP5Y0c01uElmnpq6

[246] https://github.com/curiousily/Deep-Learning-For-Hackers

[247] http://www.cis.fordham.edu/wisdm/dataset.php

[248] http://www.cis.fordham.edu/wisdm/includes/files/sensorKDD-2010.pdf

```
!gdown --id 152sWECukjvLerrVG2NUO8gtMFg83RKCF --output WISDM_ar_latest.tar.gz
!tar -xvf WISDM_ar_latest.tar.gz
```

The raw file is missing column names. Also, one of the columns is having an extra “;” after each value. Let’s fix that:

```
column_names = [
  'user_id',
  'activity',
  'timestamp',
  'x_axis',
  'y_axis',
  'z_axis'
]

df = pd.read_csv(
  'WISDM_ar_v1.1/WISDM_ar_v1.1_raw.txt',
  header=None,
  names=column_names
)

df.z_axis.replace(regex=True, inplace=True, to_replace=r';', value=r'')
df['z_axis'] = df.z_axis.astype(np.float64)
df.dropna(axis=0, how='any', inplace=True)
df.shape
```

```
(1098203, 6)
```

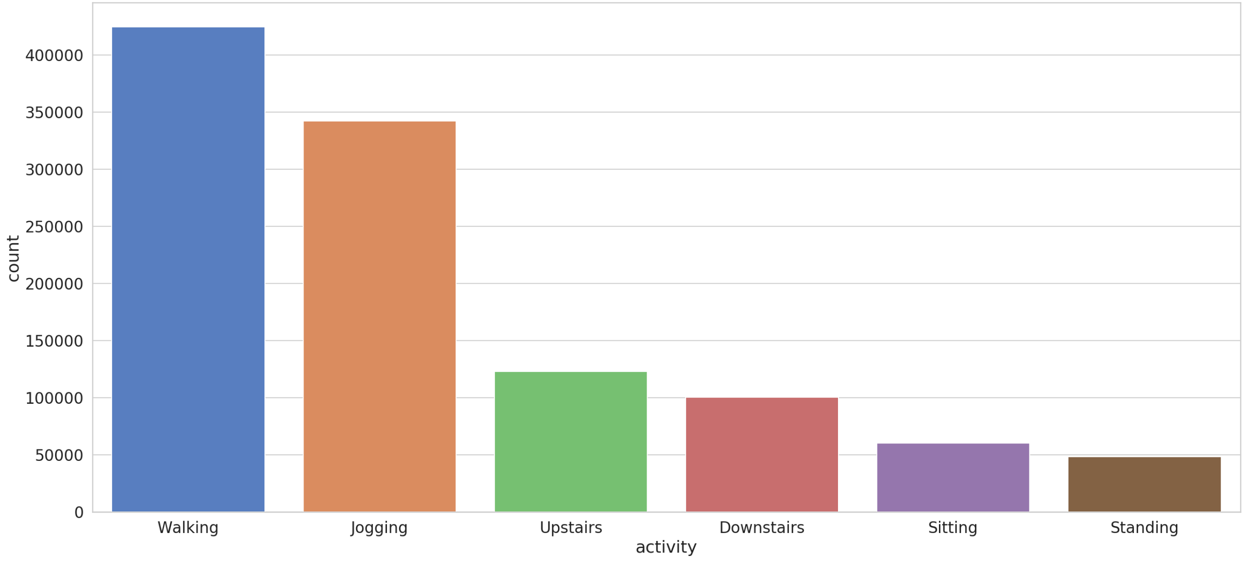
The data has the following features:

- `user_id` - unique identifier of the user doing the activity
- `activity` - the category of the current activity
- `timestamp`
- `x_axis`, `y_axis`, `z_axis` - accelerometer data for each axis

What can we learn from the data?

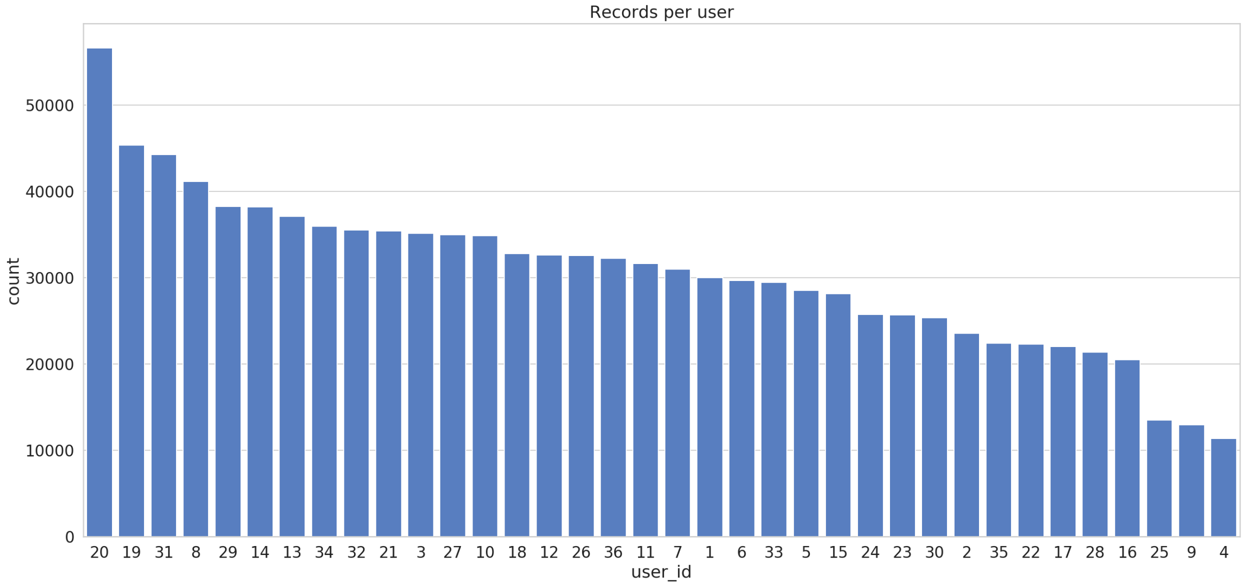
Exploration

We have six different categories. Let’s look at their distribution:



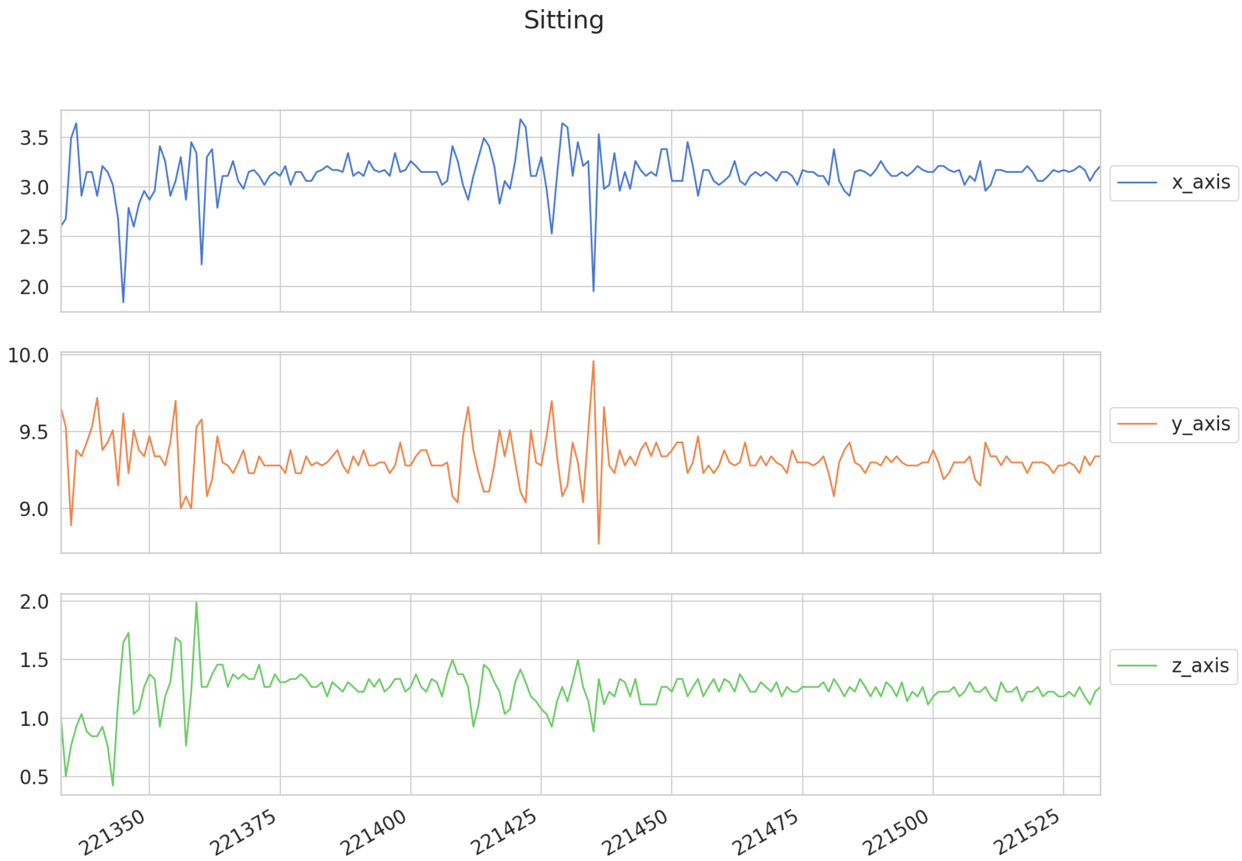
Walking and jogging are severely overrepresented. You might apply some techniques to balance the dataset.

We have multiple users. How much data do we have per user?

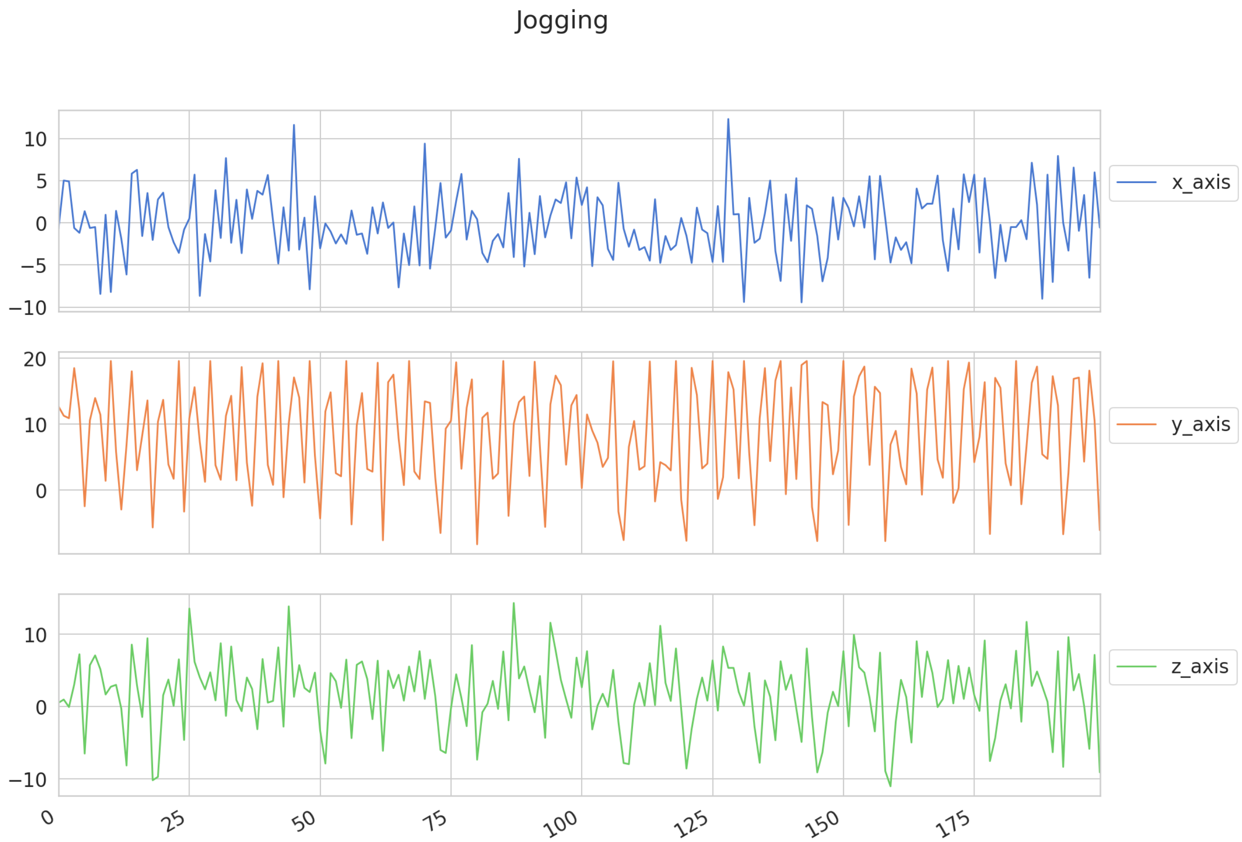


Most users (except the last 3) have a decent amount of records.

How do different types of activities look like? Let's take the first 200 records and have a look:



Sitting is well, pretty relaxed. How about jogging?



This looks much bouncier. Good, the type of activities can be separated/classified by observing the data (at least for that sample of those 2 activities).

We need to figure out a way to turn the data into sequences along with the category for each one.

Preprocessing

The first thing we need to do is to split the data into training and test datasets. We'll use the data from users with id below or equal to 30. The rest will be for training:

```
df_train = df[df['user_id'] <= 30]
df_test = df[df['user_id'] > 30]
```

Next, we'll scale the accelerometer data values:

```
scale_columns = ['x_axis', 'y_axis', 'z_axis']

scaler = RobustScaler()

scaler = scaler.fit(df_train[scale_columns])

df_train.loc[:, scale_columns] = scaler.transform(
  df_train[scale_columns].to_numpy()
)

df_test.loc[:, scale_columns] = scaler.transform(
  df_test[scale_columns].to_numpy()
)
```

Note that we fit the scaler only on the training data. How can we create the sequences? We'll just modify the `create_dataset` function a bit:

```
def create_dataset(X, y, time_steps=1, step=1):
    Xs, ys = [], []
    for i in range(0, len(X) - time_steps, step):
        v = X.iloc[i:(i + time_steps)].values
        labels = y.iloc[i: i + time_steps]
        Xs.append(v)
        ys.append(stats.mode(labels)[0][0])
    return np.array(Xs), np.array(ys).reshape(-1, 1)
```

We choose the label (category) by using the mode[249] of all categories in the sequence. That is, given a sequence of length `time_steps`, we're are classifying it as the category that occurs most often.

Here's how to create the sequences:

```
TIME_STEPS = 200
STEP = 40

X_train, y_train = create_dataset(
    df_train[['x_axis', 'y_axis', 'z_axis']],
    df_train.activity,
    TIME_STEPS,
    STEP
)

X_test, y_test = create_dataset(
```

[249] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mode.html

```
    df_test[['x_axis', 'y_axis', 'z_axis']],
    df_test.activity,
    TIME_STEPS,
    STEP
)
```

Let's have a look at the shape of the new sequences:

```
print(X_train.shape, y_train.shape)
```

```
(22454, 200, 3) (22454, 1)
```

We have significantly reduced the amount of training and test data. Let's hope that our model will still learn something useful.

The last preprocessing step is the encoding of the categories:

```
enc = OneHotEncoder(handle_unknown='ignore', sparse=False)

enc = enc.fit(y_train)

y_train = enc.transform(y_train)
y_test = enc.transform(y_test)
```

Done with the preprocessing! How good our model is going to be at recognizing user activities?

Classifying Human Activity

We'll start with a simple Bidirectional LSTM model. You can try and increase the complexity. Note that the model is relatively slow to train:

```
model = keras.Sequential()
model.add(
    keras.layers.Bidirectional(
      keras.layers.LSTM(
          units=128,
          input_shape=[X_train.shape[1], X_train.shape[2]]
      )
    )
)
model.add(keras.layers.Dropout(rate=0.5))
model.add(keras.layers.Dense(units=128, activation='relu'))
model.add(keras.layers.Dense(y_train.shape[1], activation='softmax'))

model.compile(
  loss='categorical_crossentropy',
  optimizer='adam',
  metrics=['acc']
)
```

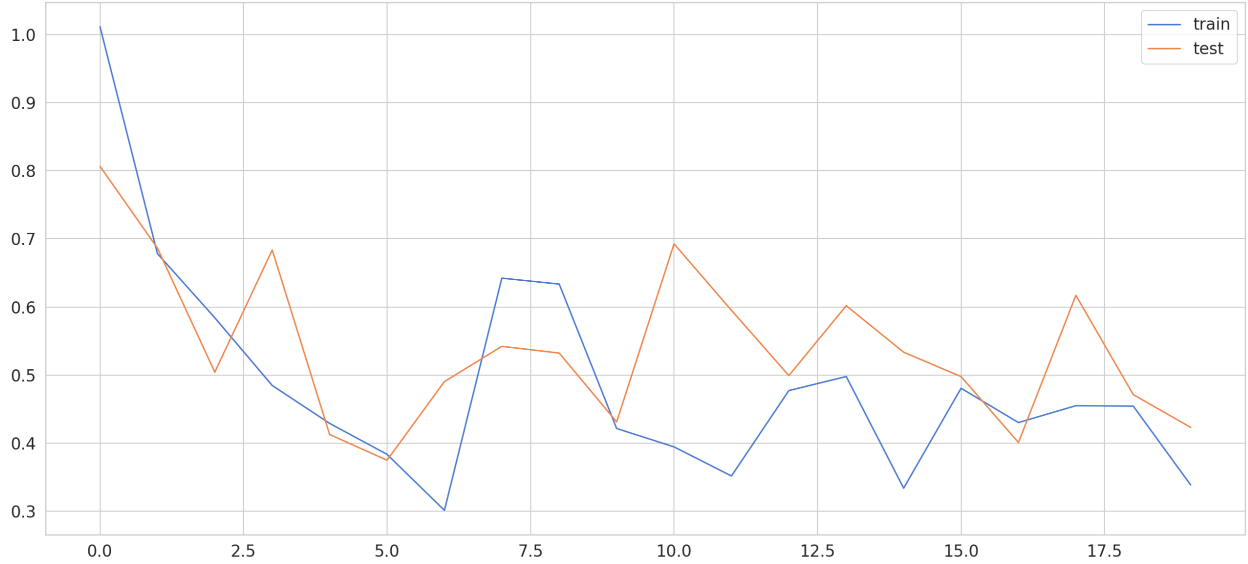
The actual training progress is straightforward (remember to not shuffle):

```
history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_split=0.1,
    shuffle=False
)
```

How good is our model?

Evaluation

Here's how the training process went:



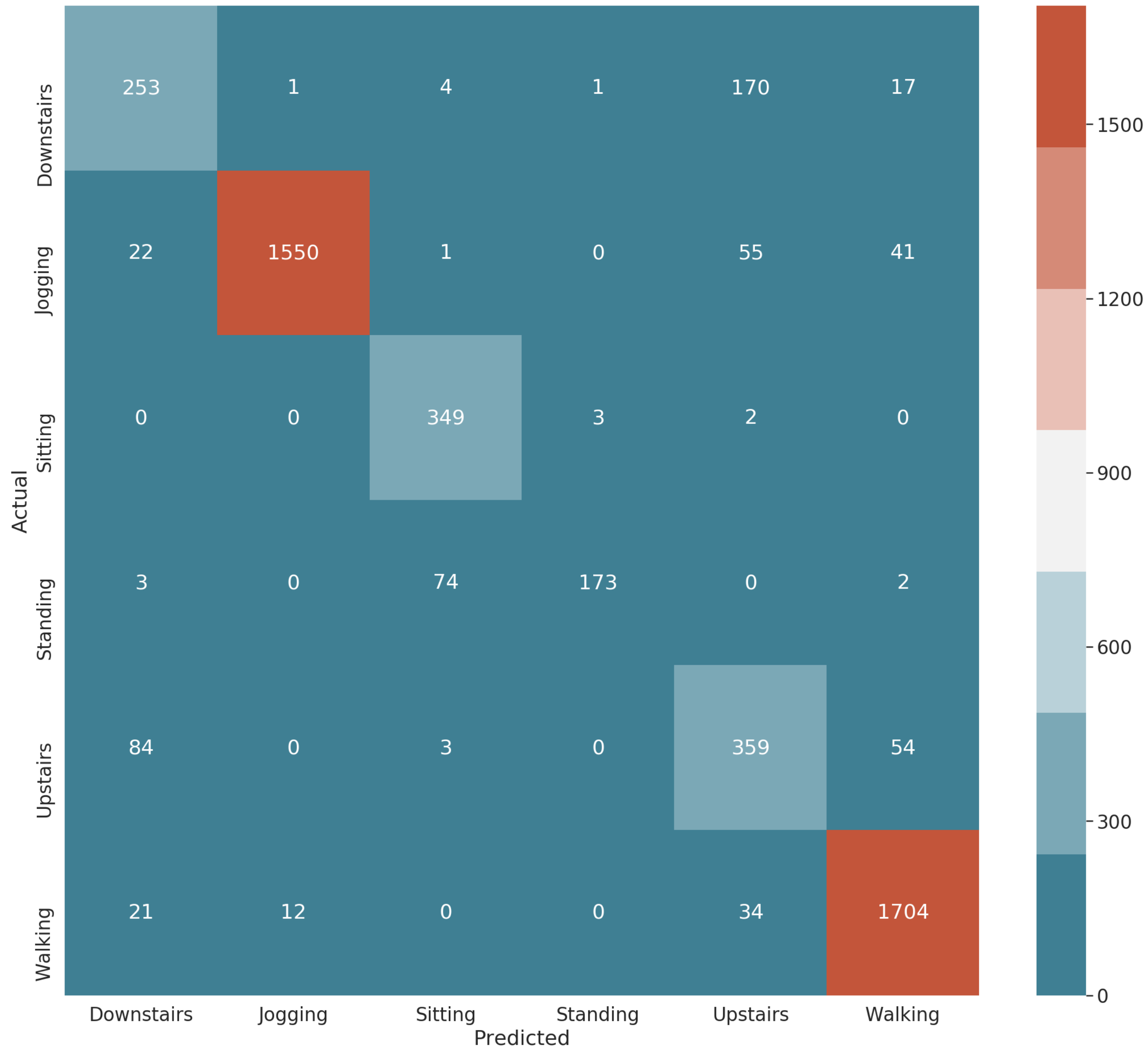
You can surely come up with a better model/hyperparameters and improve it. How well can it predict the test data?

```
model.evaluate(X_test, y_test)
```

```
[0.3619675412960649, 0.8790064]
```

~*88%* accuracy. Not bad for a quick and dirty model. Let's have a look at the confusion matrix:

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



Our model is confusing the Upstairs and Downstairs activities. That's somewhat expected. Additionally, when developing a real-world application, you might merge those two and consider them a single class/category. Recall that there is a significant imbalance in our dataset, too.

Conclusion

You did it! You've build a model that recognizes activity from 200 records of accelerometer data. Your model achieves $\sim$*88%* accuracy on the test data. Here are the steps you took:

- Load Human Activity Recognition Data
- Build LSTM Model for Classification
- Evaluate the Model

You learned how to build a Bidirectional LSTM model and classify Time Series data. There is even more fun with LSTMs and Time Series coming next :)

Run the complete notebook in your browser[250]

The complete project on GitHub[251]

References

- TensorFlow - Time series forecasting[252]
- Understanding LSTM Networks[253]
- WISDM: WIreless Sensor Data Mining[254]

[250] https://colab.research.google.com/drive/1hxq4-A4SZYfKqmqfwP5Y0c01uElmnpq6
[251] https://github.com/curiousily/Deep-Learning-For-Hackers
[252] https://www.tensorflow.org/tutorials/structured_data/time_series
[253] https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[254] http://www.cis.fordham.edu/wisdm/dataset.php

Time Series Anomaly Detection with LSTM Autoencoders using Keras in Python

> TL;DR Detect anomalies in S&P 500 daily closing price. Build LSTM Autoencoder Neural Net for anomaly detection using Keras and TensorFlow 2.

This guide will show you how to build an Anomaly Detection model for Time Series data. You'll learn how to use LSTMs and Autoencoders in Keras and TensorFlow 2. We'll use the model to find anomalies in S&P 500 daily closing prices.

This is the plan:

- Anomaly Detection
- LSTM Autoencoders
- S&P 500 Index Data
- LSTM Autoencoder in Keras
- Finding Anomalies

Run the complete notebook in your browser[255]

The complete project on GitHub[256]

Anomaly Detection

Anomaly detection[257] refers to the task of finding/identifying rare events/data points. Some applications include - bank fraud detection, tumor detection in medical imaging, and errors in written text.

A lot of supervised and unsupervised approaches to anomaly detection has been proposed. Some of the approaches include - One-class SVMs, Bayesian Networks, Cluster analysis, and (of course) Neural Networks.

We will use an LSTM Autoencoder Neural Network to detect/predict anomalies (sudden price changes) in the S&P 500 index.

[255] https://colab.research.google.com/drive/1MrBsc03YLYN81qAhFGToIFRMDoh3MAoM
[256] https://github.com/curiousily/Deep-Learning-For-Hackers
[257] https://en.wikipedia.org/wiki/Anomaly_detection

LSTM Autoencoders

Autoencoders Neural Networks[258] try to learn data representation of its input. So the input of the Autoencoder is the same as the output? Not quite. Usually, we want to learn an efficient encoding that uses fewer parameters/memory.

The encoding should allow for output similar to the original input. In a sense, we're forcing the model to learn the most important features of the data using as few parameters as possible.

Anomaly Detection with Autoencoders

Here are the basic steps to Anomaly Detection using an Autoencoder:

1. Train an Autoencoder on normal data (no anomalies)
2. Take a new data point and try to reconstruct it using the Autoencoder
3. If the error (reconstruction error) for the new data point is above some threshold, we label the example as an anomaly

Good, but is this useful for Time Series Data? Yes, we need to take into account the temporal properties of the data. Luckily, LSTMs can help us with that.

S&P 500 Index Data

Our data is the daily closing prices for the S&P 500 index from 1986 to 2018.

> The S&P 500, or just the S&P, is a stock market index that measures the stock performance of 500 large companies listed on stock exchanges in the United States. It is one of the most commonly followed equity indices, and many consider it to be one of the best representations of the U.S. stock market. -Wikipedia[259]

It is provided by Patrick David[260] and hosted on Kaggle[261]. The data contains only two columns/features - the date and the closing price. Let's download and load into a Data Frame:

```
!gdown --id 10vdMg_RazoIatwrT7azKFX4P02OebU76 --output spx.csv
```

[258] https://en.wikipedia.org/wiki/Autoencoder

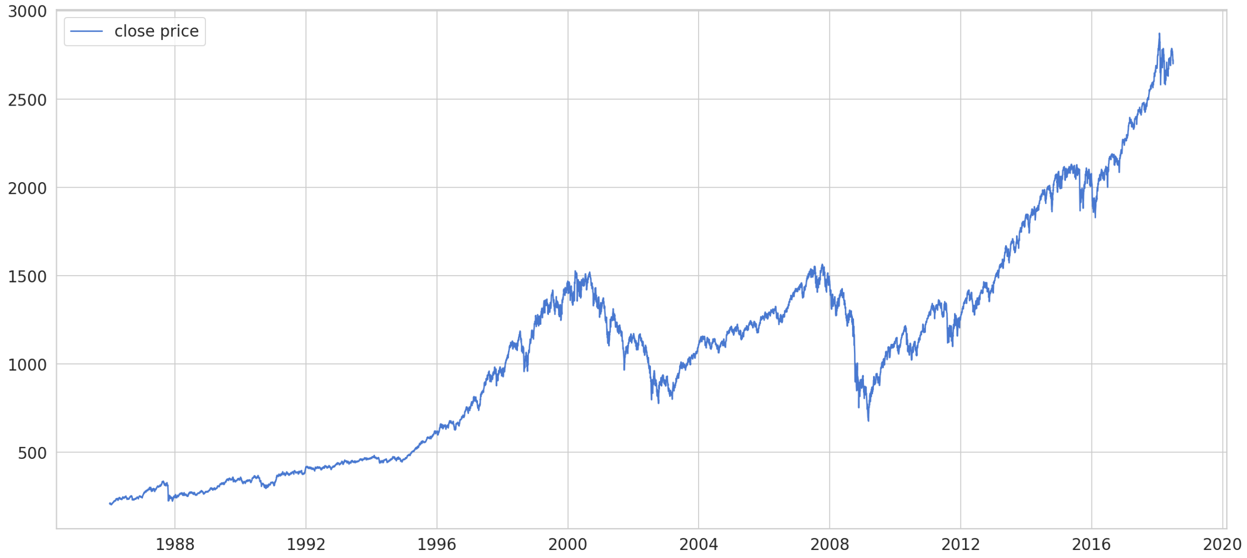
[259] https://en.wikipedia.org/wiki/S%26P_500_Index

[260] https://twitter.com/pdquant

[261] https://www.kaggle.com/pdquant/sp500-daily-19862018

```
df = pd.read_csv('spx.csv', parse_dates=['date'], index_col='date')
```

Let's have a look at the daily close price:



That trend (last 8 or so years) looks really juicy. You might want to board the train. When should you buy or sell? How early can you "catch" sudden changes/anomalies?

Preprocessing

We'll use 95% of the data and train our model on it:

```
train_size = int(len(df) * 0.95)
test_size = len(df) - train_size
train, test = df.iloc[0:train_size], df.iloc[train_size:len(df)]
print(train.shape, test.shape)
```

```
(7782, 1) (410, 1)
```

Next, we'll rescale the data using the training data and apply the same transformation to the test data:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler = scaler.fit(train[['close']])

train['close'] = scaler.transform(train[['close']])
test['close'] = scaler.transform(test[['close']])
```

Finally, we'll split the data into subsequences. Here's the little helper function for that:

```
def create_dataset(X, y, time_steps=1):
    Xs, ys = [], []
    for i in range(len(X) - time_steps):
        v = X.iloc[i:(i + time_steps)].values
        Xs.append(v)
        ys.append(y.iloc[i + time_steps])
    return np.array(Xs), np.array(ys)
```

We'll create sequences with 30 days worth of historical data:

```
TIME_STEPS = 30

## reshape to [samples, time_steps, n_features]

X_train, y_train = create_dataset(
  train[['close']],
  train.close,
  TIME_STEPS
)

X_test, y_test = create_dataset(
  test[['close']],
  test.close,
  TIME_STEPS
)

print(X_train.shape)
```

```
(7752, 30, 1)
```

The shape of the data looks correct. How can we make LSTM Autoencoder in Keras?

LSTM Autoencoder in Keras

Our Autoencoder should take a sequence as input and outputs a sequence of the same shape. Here's how to build such a simple model in Keras:

```
model = keras.Sequential()
model.add(keras.layers.LSTM(
    units=64,
    input_shape=(X_train.shape[1], X_train.shape[2])
))
model.add(keras.layers.Dropout(rate=0.2))
model.add(keras.layers.RepeatVector(n=X_train.shape[1]))
model.add(keras.layers.LSTM(units=64, return_sequences=True))
model.add(keras.layers.Dropout(rate=0.2))
model.add(
  keras.layers.TimeDistributed(
    keras.layers.Dense(units=X_train.shape[2])
  )
)

model.compile(loss='mae', optimizer='adam')
```

There are a couple of things that might be new to you in this model. The RepeatVector[262] layer simply repeats the input *n* times. Adding `return_sequences=True` in LSTM layer makes it return the sequence.

Finally, the TimeDistributed[263] layer creates a vector with a length of the number of outputs from the previous layer. Your first LSTM Autoencoder is ready for training.

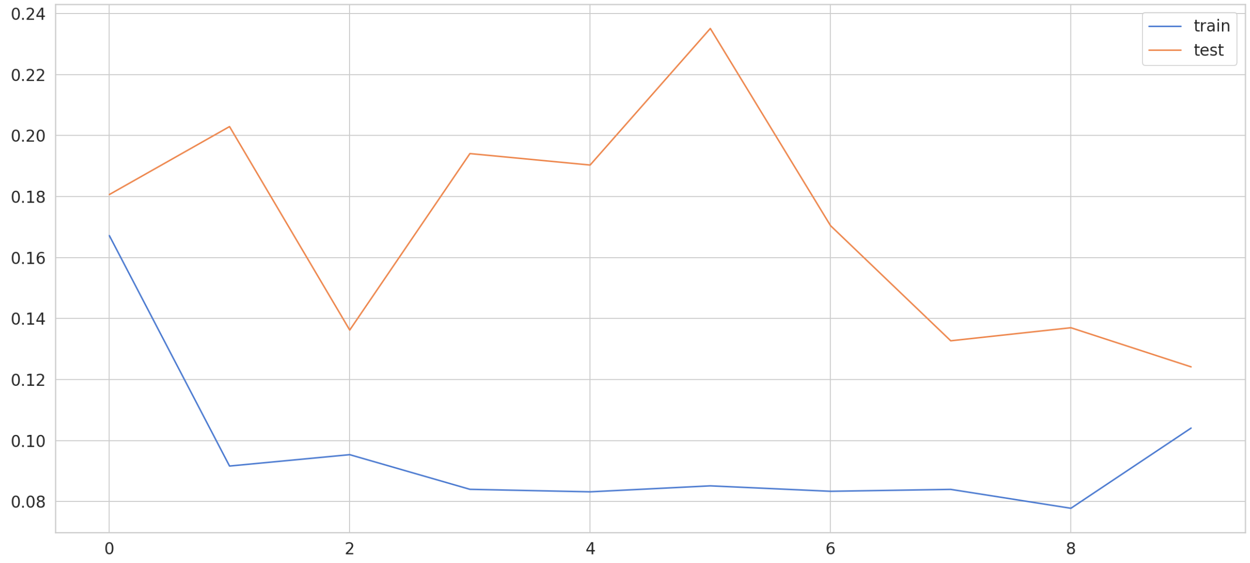
Training the model is no different from a regular LSTM model:

```
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=32,
    validation_split=0.1,
    shuffle=False
)
```

[262] https://www.tensorflow.org/api_docs/python/tf/keras/layers/RepeatVector
[263] https://www.tensorflow.org/api_docs/python/tf/keras/layers/TimeDistributed

Evaluation

We've trained our model for 10 epochs with less than 8k examples. Here are the results:

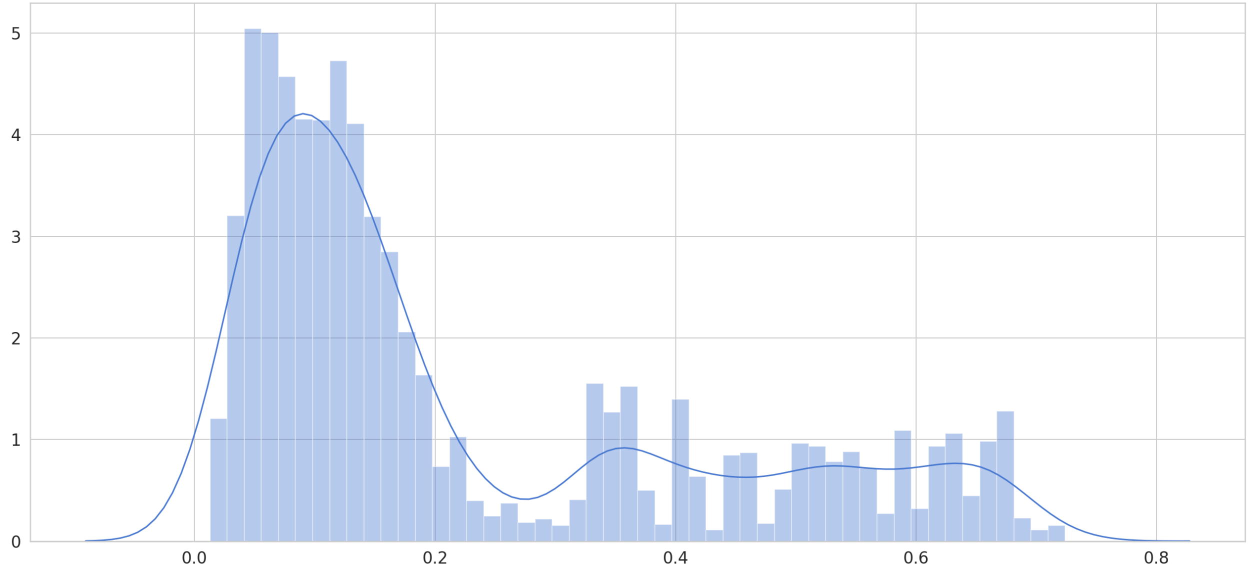


Finding Anomalies

Still, we need to detect anomalies. Let's start with calculating the Mean Absolute Error (MAE) on the training data:

```
X_train_pred = model.predict(X_train)

train_mae_loss = np.mean(np.abs(X_train_pred - X_train), axis=1)
```

Let's have a look at the error:



We'll pick a threshold of 0.65, as not much of the loss is larger than that. When the error is larger than that, we'll declare that example an anomaly:

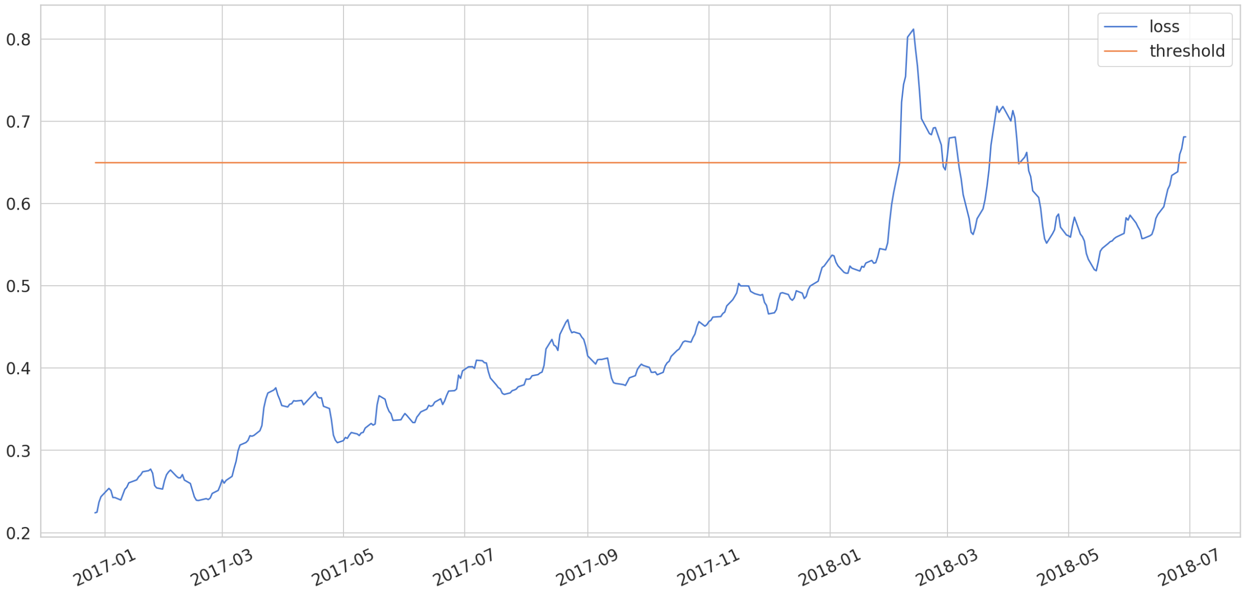
```
THRESHOLD = 0.65
```

Let's calculate the MAE on the test data:

```
X_test_pred = model.predict(X_test)

test_mae_loss = np.mean(np.abs(X_test_pred - X_test), axis=1)
```

We'll build a DataFrame containing the loss and the anomalies (values above the threshold):

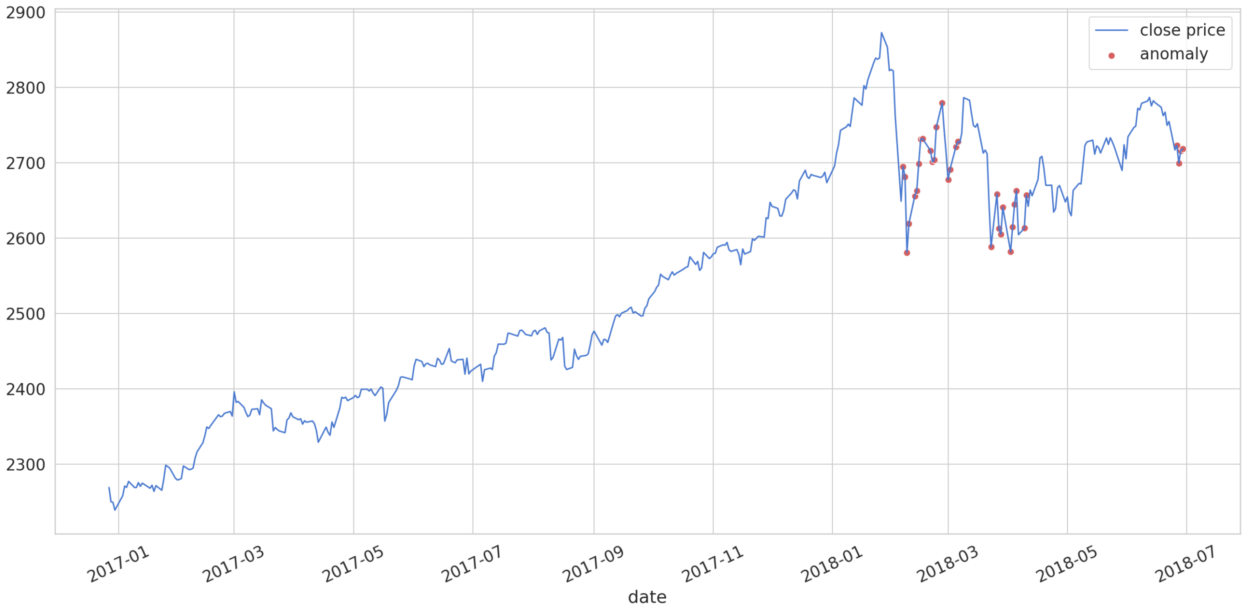
```
test_score_df = pd.DataFrame(index=test[TIME_STEPS:].index)
test_score_df['loss'] = test_mae_loss
test_score_df['threshold'] = THRESHOLD
test_score_df['anomaly'] = test_score_df.loss > test_score_df.threshold
test_score_df['close'] = test[TIME_STEPS:].close
```



Looks like we're thresholding extreme values quite well. Let's create a DataFrame using only those:

```
anomalies = test_score_df[test_score_df.anomaly == True]
```

Finally, let's look at the anomalies found in the testing data:



You should have a thorough look at the chart. The red dots (anomalies) are covering most of the points with abrupt changes to the closing price. You can play around with the threshold and try to get even better results.

Conclusion

You just combined two powerful concepts in Deep Learning - LSTMs and Autoencoders. The result is a model that can find anomalies in S&P 500 closing price data. You can try to tune the model and/or the threshold to get even better results.

Here's a recap of what you did:

- Anomaly Detection
- LSTM Autoencoders
- S&P 500 Index Data
- LSTM Autoencoder in Keras
- Finding Anomalies

Run the complete notebook in your browser[264]

The complete project on GitHub[265]

Can you apply the model to your dataset? What results did you get?

References

- TensorFlow - Time series forecasting[266]
- Understanding LSTM Networks[267]
- Step-by-step understanding LSTM Autoencoder layers[268]
- S&P500 Daily Prices 1986 - 2018[269]

[264] https://colab.research.google.com/drive/1MrBsc03YLYN81qAhFGToIFRMDoh3MAoM
[265] https://github.com/curiousily/Deep-Learning-For-Hackers
[266] https://www.tensorflow.org/tutorials/structured_data/time_series
[267] https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[268] https://towardsdatascience.com/step-by-step-understanding-lstm-autoencoder-layers-ffab055b6352
[269] https://www.kaggle.com/pdquant/sp500-daily-19862018

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[270]

The complete project on GitHub[271]

Object Detection

Object detection[272] 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.

[270] https://colab.research.google.com/drive/1ldnii3sGJaUHPV6TWImykbeE_O-8VIIN

[271] https://github.com/curiousily/Deep-Learning-For-Hackers

[272] 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)[273], 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.

[273] 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[274]. 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[275] 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[276] 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
    }
```

[274] https://paperswithcode.com/sota/object-detection-on-coco
[275] https://github.com/fizyr/keras-retinanet
[276] 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)
```

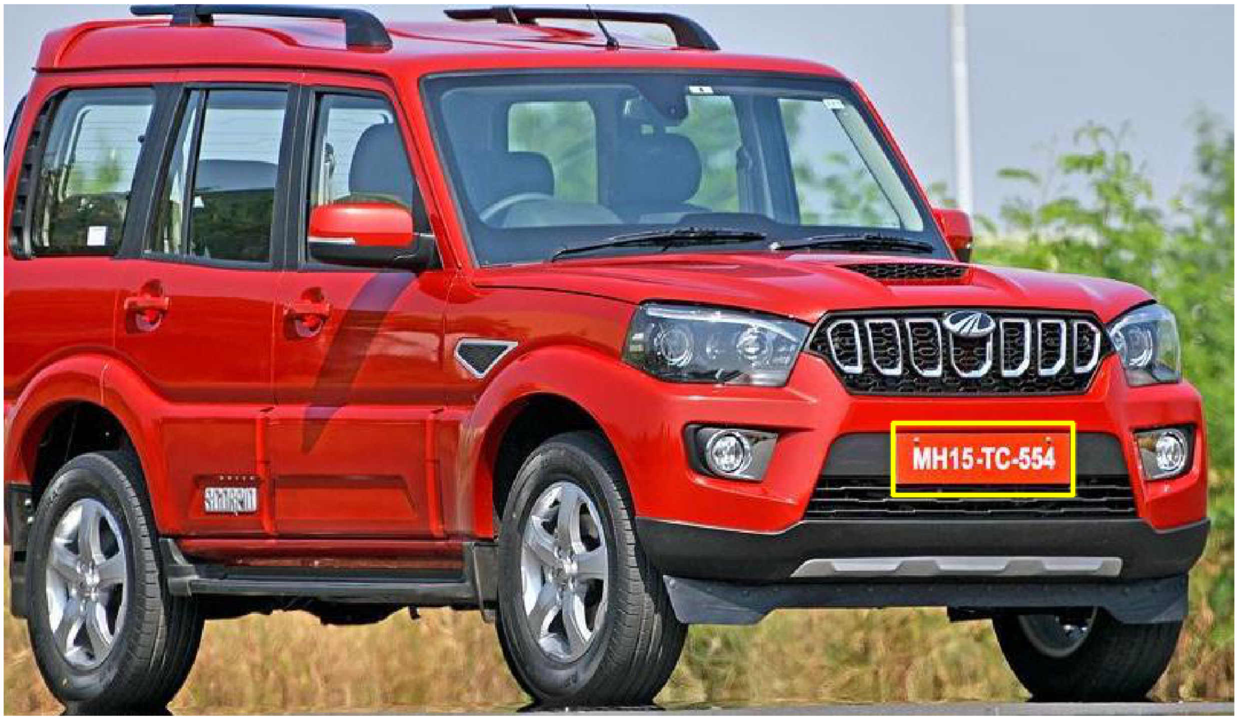
We can unify the download and the creation of annotation file like so:

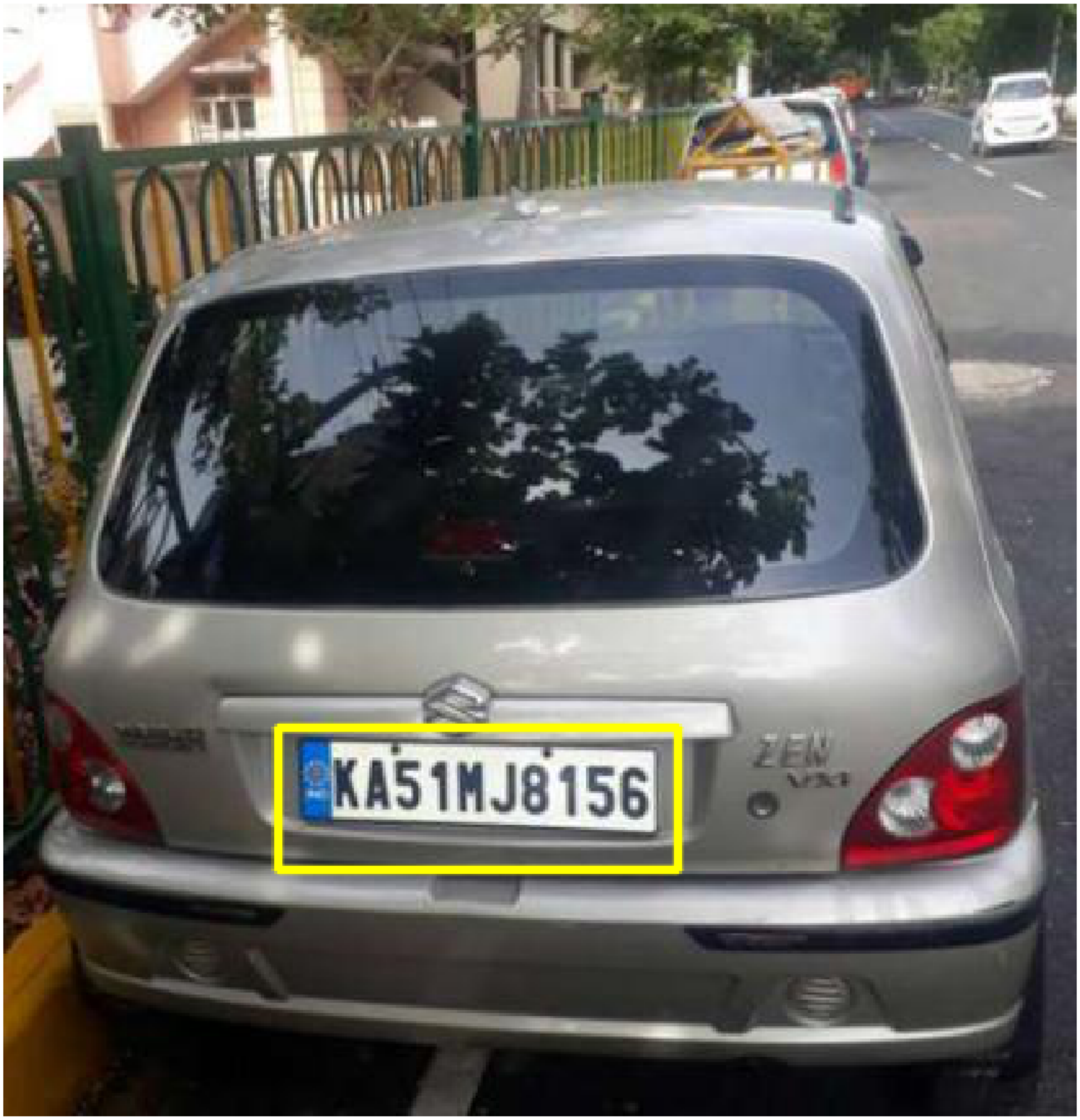
```
dataset = dict()
dataset["image_name"] = list()
dataset["top_x"] = list()
dataset["top_y"] = list()
dataset["bottom_x"] = list()
dataset["bottom_y"] = list()
dataset["class_name"] = list()

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:

```
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:

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

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

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

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

```
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:

```
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[277] 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:

[277] 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:

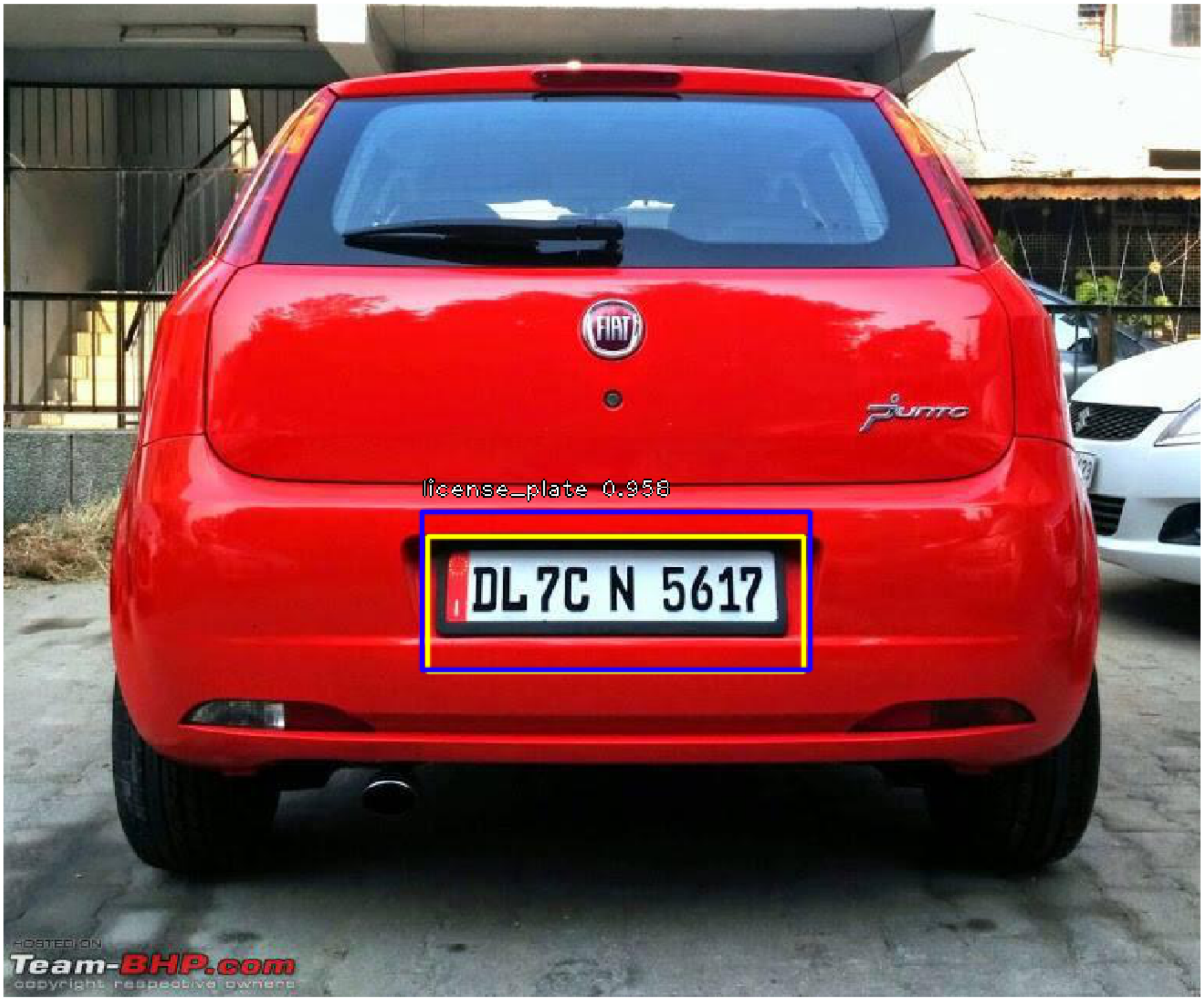
```
THRES_SCORE = 0.6

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)
```

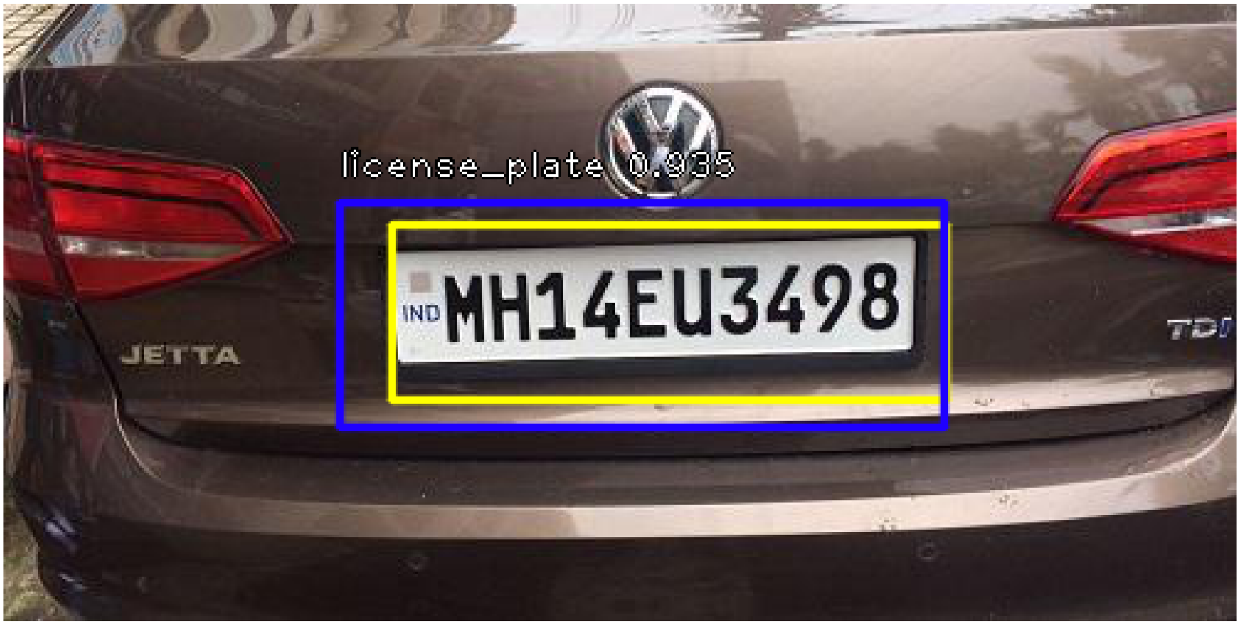
We'll draw detections with a confidence score above 0.6. Note that the scores are sorted high to low, so breaking from the loop is fine.

Let's put everything together:

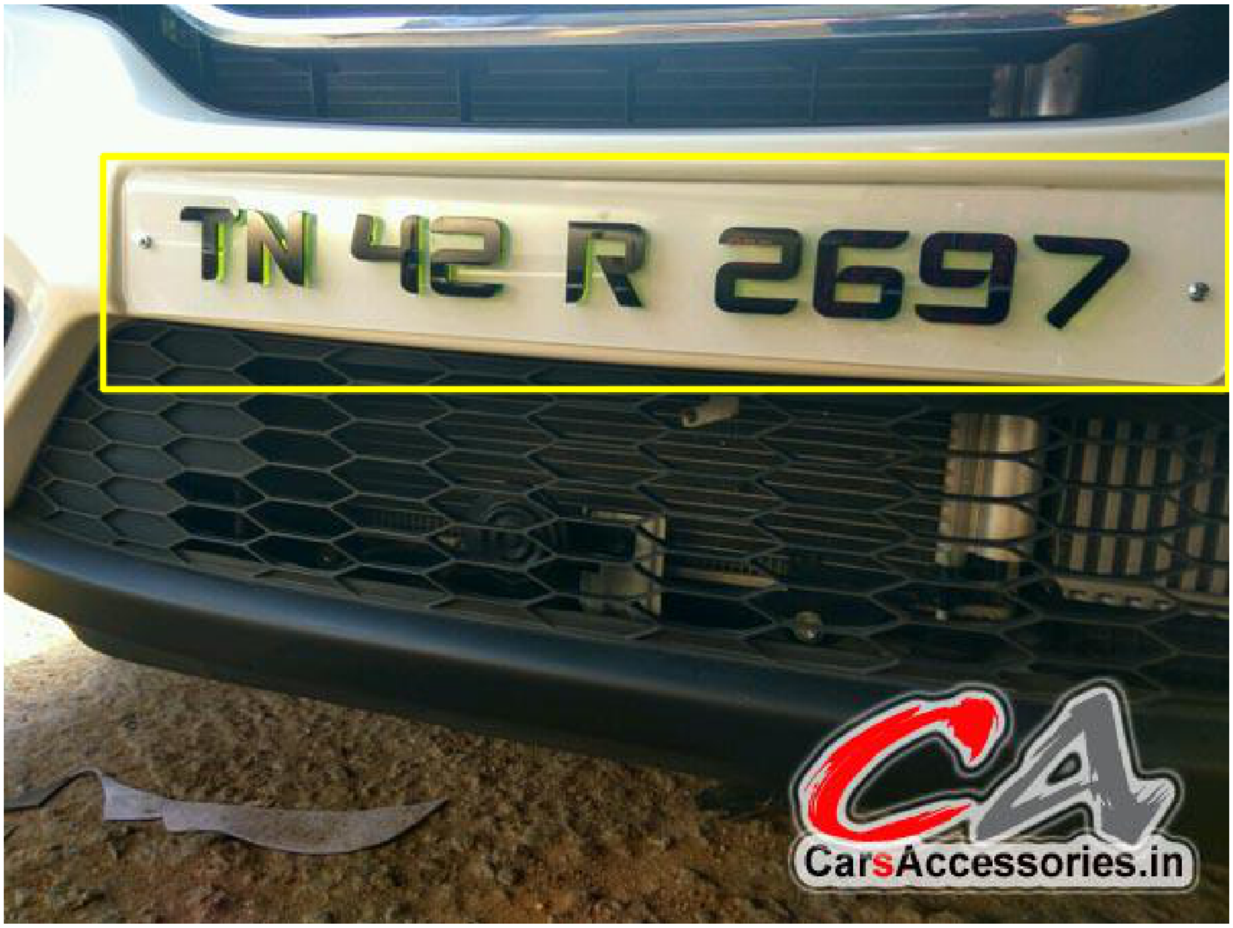
```
def show_detected_objects(image_row):
  img_path = image_row.image_name

  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[278]

The complete project on GitHub[279]

References

- Keras RetinaNet[280]
- Vehicle Number Plate Detection[281]
- Object detection: speed and accuracy comparison[282]
- Focal Loss for Dense Object Detection[283]
- Plate Detection –> Preparing the data[284]
- Object Detection in Colab with Fizyr Retinanet[285]

[278] https://colab.research.google.com/drive/1ldnii3sGJaUHPV6TWImykbeE_O-8VIIN
[279] https://github.com/curiousily/Deep-Learning-For-Hackers
[280] https://github.com/fizyr/keras-retinanet
[281] https://www.kaggle.com/dataturks/vehicle-number-plate-detection
[282] https://medium.com/@jonathan_hui/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359
[283] https://arxiv.org/abs/1708.02002
[284] https://www.kaggle.com/dsousa/plate-detection-preparing-the-data
[285] https://www.freecodecamp.org/news/object-detection-in-colab-with-fizyr-retinanet-efed36ac4af3/

Image Data Augmentation

> TL;DR Learn how to create new examples for your dataset using image augmentation techniques. Load a scanned document image and apply various augmentations. Create an augmented dataset for Object Detection.

Your Deep Learning models are dumb. Detecting objects in a slightly different image, compared to the training examples, can produce hugely incorrect predictions. How can you fix that?

Ideally, you would go and get more training data, and then some more. The more diverse the examples, the better. Except, getting new data can be hard, expensive, or just impossible. What can you do?

You can use your own "creativity" and create new images from the existing ones. The goal is to create transformations that resemble real examples not found in the data.

We're going to have a look at "basic" image augmentation techniques. Advanced methods like Neural Style Transfer and GAN data augmentation may provide even more performance improvements, but are not covered here.

You'll learn how to:

- Load images using OpenCV
- Apply various image augmentations
- Compose complex augmentations to simulate real-world data
- Create augmented dataset ready to use for Object Detection

Run the complete notebook in your browser[286]

The complete project on GitHub[287]

Tools for Image Augmentation

Image augmentation is widely used in practice. Your favorite Deep Learning library probably offers some tools for it.

TensorFlow 2 (Keras) gives the ImageDataGenerator[288]. PyTorch offers a much better interface via Torchvision Transforms[289]. Yet, image augmentation is a preprocessing step (you are preparing your

[286] https://colab.research.google.com/drive/12r6e0grdtssEjxYAMSAnwJj3y7fVfn-V

[287] https://github.com/curiousily/Deep-Learning-For-Hackers

[288] https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator?version=stable

[289] https://pytorch.org/docs/stable/torchvision/transforms.html

dataset for training). Experimenting with different models and frameworks means that you'll have to switch a lot of code around.

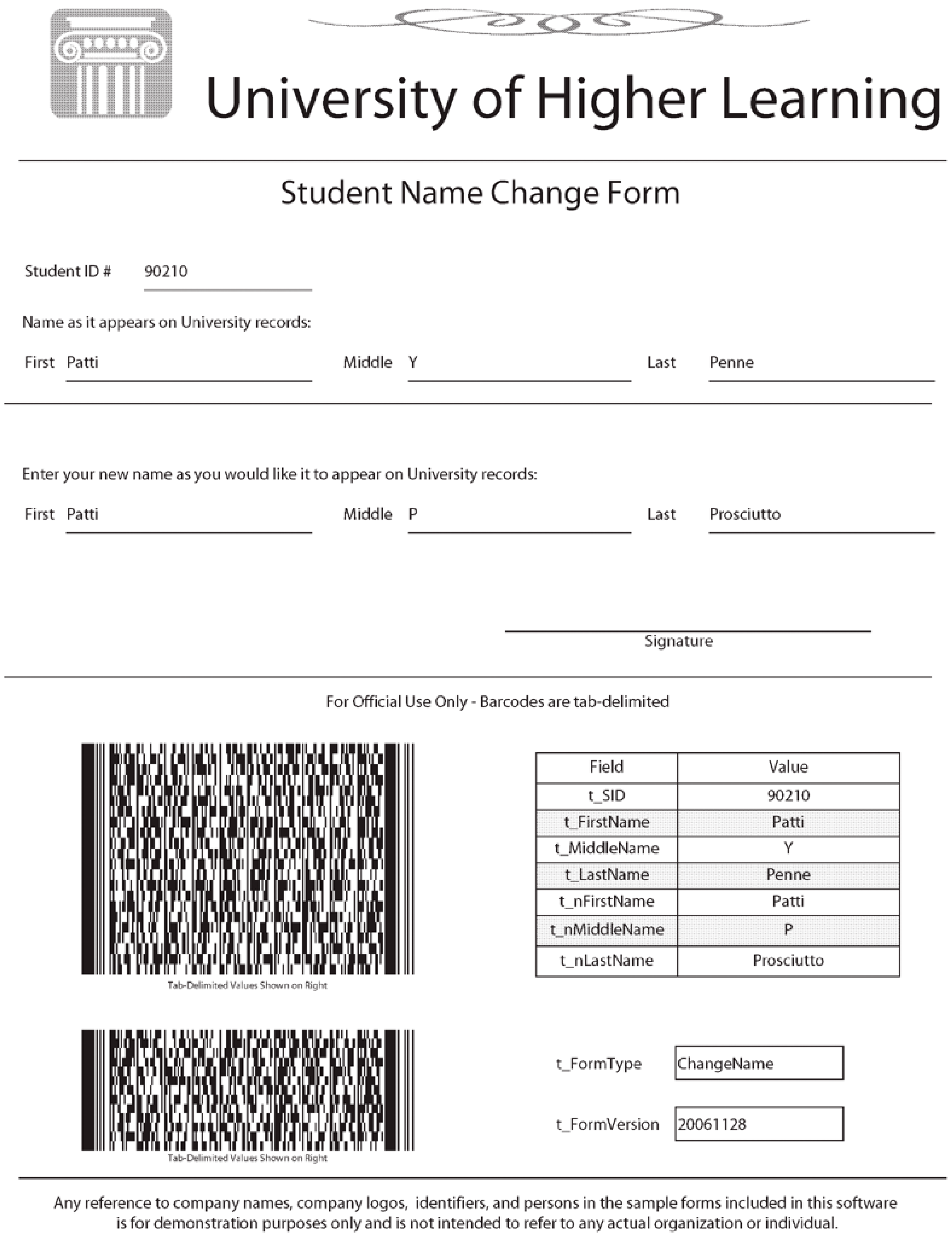
Luckily, Albumentations[290] offers a clean and easy to use API. It is independent of other Deep Learning libraries and quite fast. Also, it gives you a large number of useful transforms.

How can we use it to transform some images?

Augmenting Scanned Documents

Here is the sample scanned document, that we'll transform using Albumentations:

[290] https://github.com/albumentations-team/albumentations



University of Higher Learning

Student Name Change Form

Student ID # 90210

Name as it appears on University records:

First Patti Middle Y Last Penne

Enter your new name as you would like it to appear on University records:

First Patti Middle P Last Prosciutto

Signature

For Official Use Only - Barcodes are tab-delimited

Tab-Delimited Values Shown on Right

Field	Value
t_SID	90210
t_FirstName	Patti
t_MiddleName	Y
t_LastName	Penne
t_nFirstName	Patti
t_nMiddleName	P
t_nLastName	Prosciutto

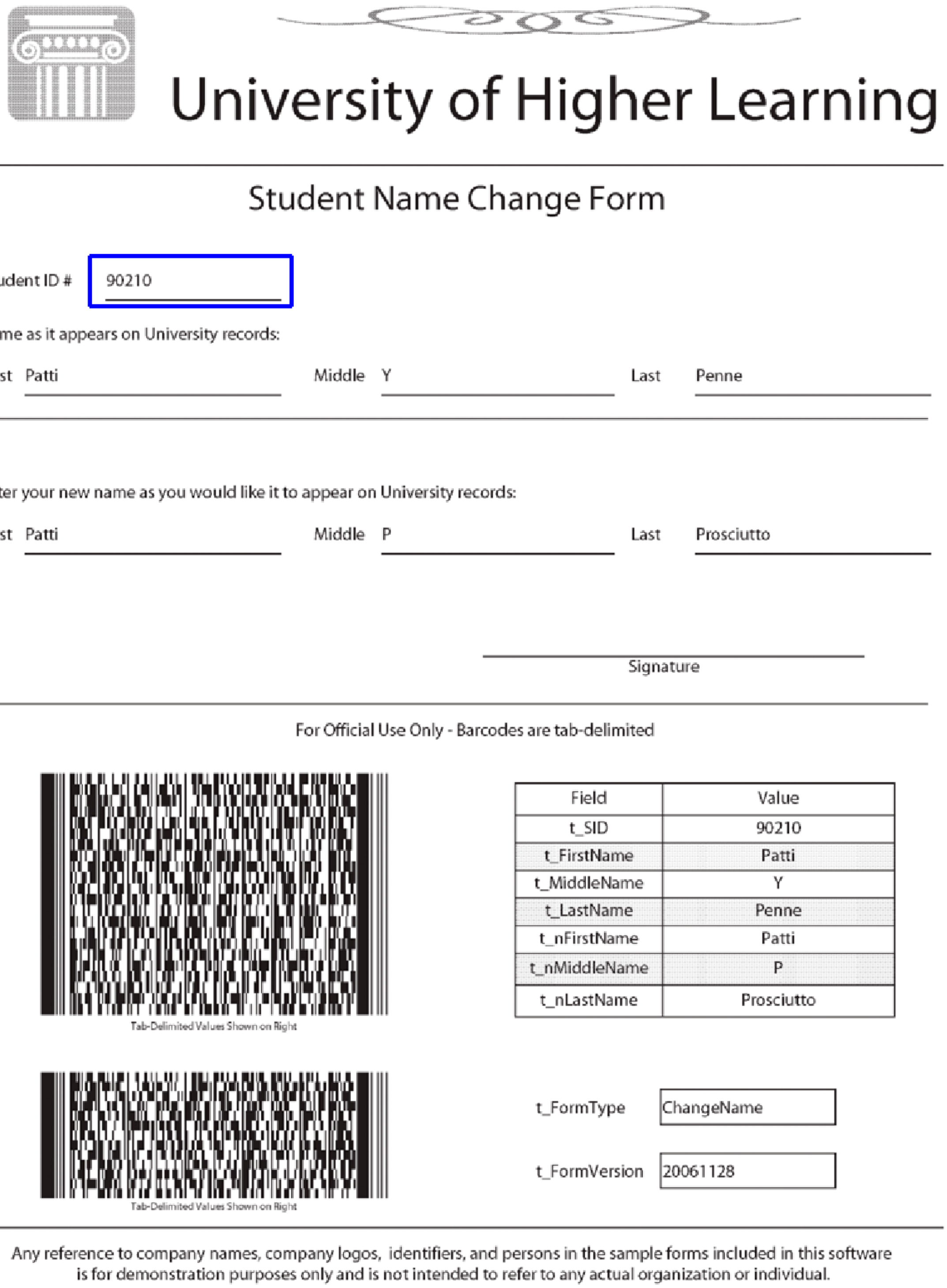
Tab-Delimited Values Shown on Right

t_FormType	ChangeName
t_FormVersion	20061128

Any reference to company names, company logos, identifiers, and persons in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization or individual.

Let’s say that you were tasked with the extraction of the Student Id from scanned documents. One way to approach the problem is to first detect the region that contains the student id and then use OCR to extract the value.

Here is the training example for our Object Detection algorithm:



University of Higher Learning

Student Name Change Form

Student ID # 90210

Name as it appears on University records:

First Patti Middle Y Last Penne

Enter your new name as you would like it to appear on University records:

First Patti Middle P Last Prosciutto

Signature

For Official Use Only - Barcodes are tab-delimited

Tab-Delimited Values Shown on Right

Field	Value
t_SID	90210
t_FirstName	Patti
t_MiddleName	Y
t_LastName	Penne
t_nFirstName	Patti
t_nMiddleName	P
t_nLastName	Prosciutto

Tab-Delimited Values Shown on Right

t_FormType ChangeName

t_FormVersion 20061128

Any reference to company names, company logos, identifiers, and persons in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization or individual.

Let's start with some basic transforms. But first, let's create some helper functions that show the augmented results:

```
def show_augmented(augmentation, image, bbox):
  augmented = augmentation(image=image, bboxes=[bbox], field_id=['1'])
  show_image(augmented['image'], augmented['bboxes'][0])
```

`show_augmented()` applies the augmentation on the image and show the result along with the modified bounding box (courtesy of Albumentations). Here is the definition of `show_image()`:

```
def show_image(image, bbox):
  image = visualize_bbox(image.copy(), bbox)
  f = plt.figure(figsize=(18, 12))
  plt.imshow(
    cv2.cvtColor(image, cv2.COLOR_BGR2RGB),
    interpolation='nearest'
  )
  plt.axis('off')
  f.tight_layout()
  plt.show()
```

We start by drawing the bounding box on top of the image and showing the result. Note that OpenCV2 uses a different channel ordering than the standard RGB. We take care of that, too.

Finally, the definition of `visualize_bbox()`:

```
BOX_COLOR = (255, 0, 0)

def visualize_bbox(img, bbox, color=BOX_COLOR, thickness=2):
    x_min, y_min, x_max, y_max = map(lambda v: int(v), bbox)

    cv2.rectangle(
      img,
      (x_min, y_min),
      (x_max, y_max),
      color=color,
      thickness=thickness
    )
    return img
```

Bounding boxes are just rectangles drawn on top of the image. We use OpenCV's `rectangle()` function and specify the top-left and bottom-right points.

Augmenting bounding boxes requires a specification of the coordinates format:

```
## [x_min, y_min, x_max, y_max], e.g. [97, 12, 247, 212].

bbox_params = A.BboxParams(
  format='pascal_voc',
  min_area=1,
  min_visibility=0.5,
  label_fields=['field_id']
)
```

Let's do some image augmentation!

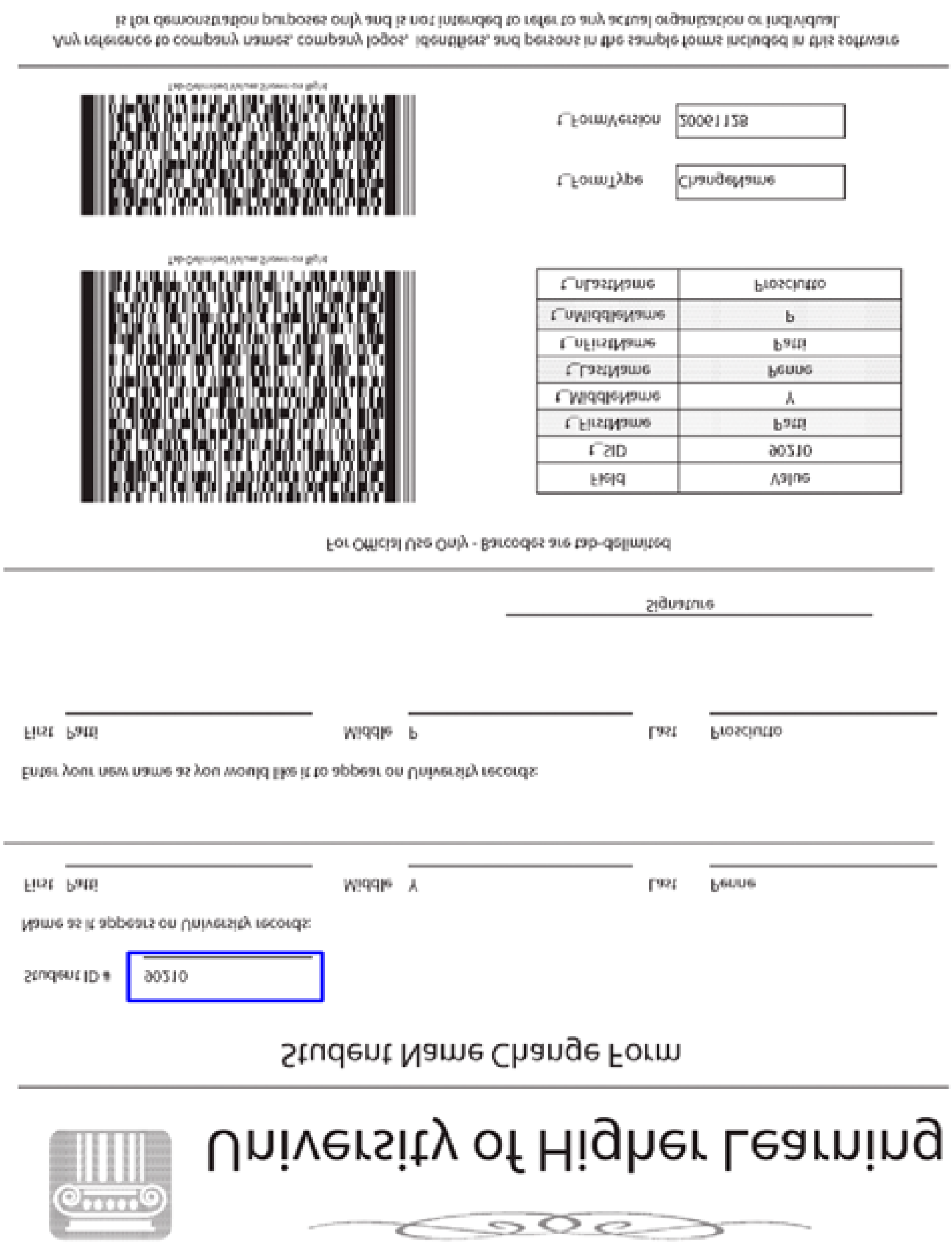
Applying Transforms

Ever worked with scanned documents? If you did, you'll know that two of the most common scanning mistakes that users make are flipping and rotation of the documents.

> Applying an augmentation multiple times will result in a different result (depending on the augmentation and parameters)

Let's start with a flip augmentation:

```
aug = A.Compose([
    A.Flip(always_apply=True)
], bbox_params=bbox_params)
```



University of Higher Learning

Student Name Change Form

Student ID # 90210

Name as it appears on University records:

First Patti Middle Y Last Penne

Enter your new name as you would like it to appear on University records:

First Patti Middle P Last Prosciutto

Signature

For Official Use Only - Barcodes are tab-delimited

Field	Value
t_SID	90210
t_FirstName	Patti
t_MiddleName	Y
t_LastName	Penne
t_nFirstName	Patti
t_nMiddleName	P
t_nLastName	Prosciutto

Tab-Delimited Values Shown on Right

t_FormType ChangeName

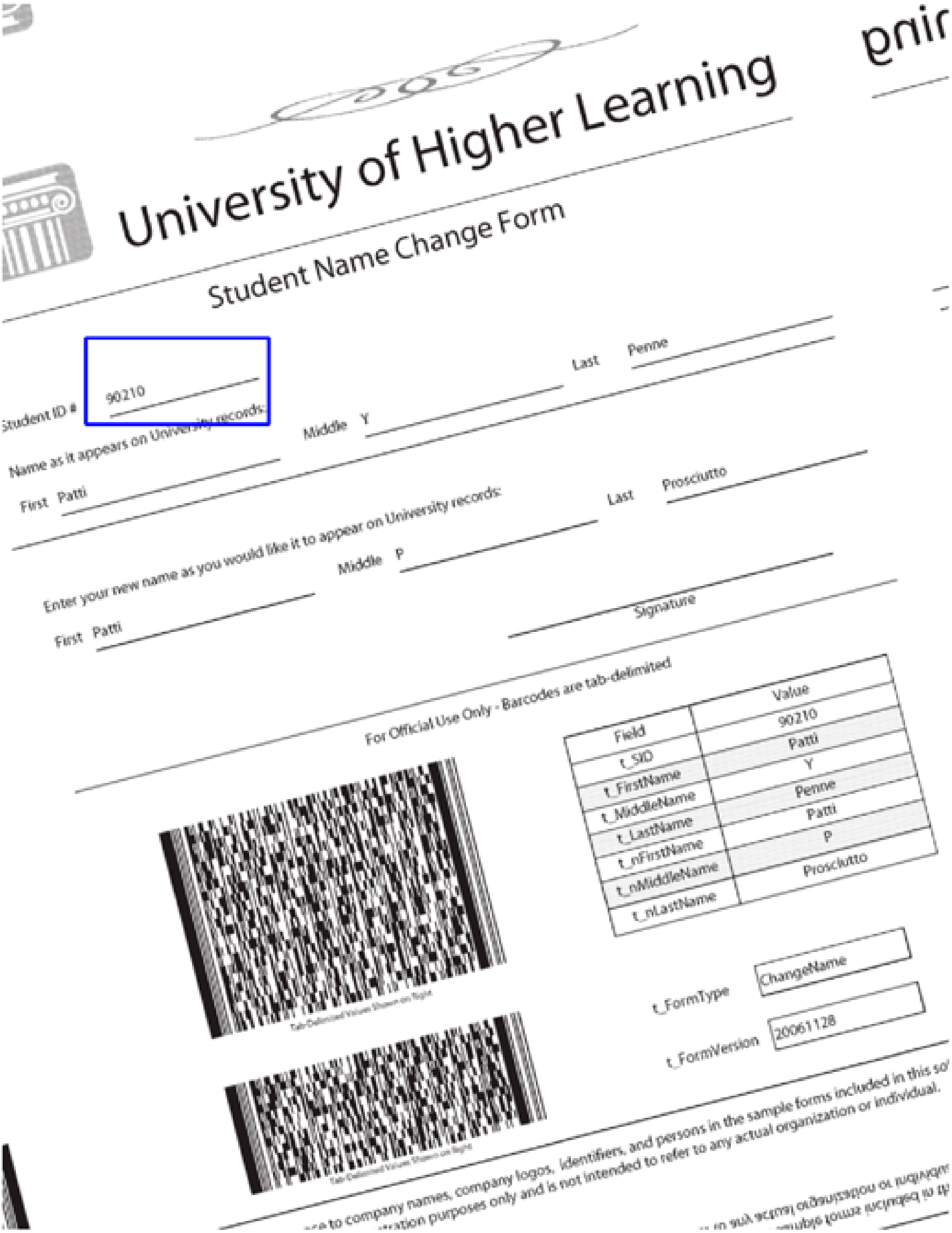
t_FormVersion 20061128

Tab-Delimited Values Shown on Right

Any reference to company names, company logos, identifiers, and persons in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization or individual.

and rotate:

```
aug = A.Compose([
  A.Rotate(limit=80, always_apply=True)
], bbox_params=bbox_params)
```



University of Higher Learning

Student Name Change Form

Student ID # 90210

Name as it appears on University records:

First Patti Middle Y Last Penne

Enter your new name as you would like it to appear on University records:

First Patti Middle P Last Prosciutto

Signature

For Official Use Only - Barcodes are tab-delimited

Field	Value
t_SID	90210
t_FirstName	Patti
t_MiddleName	Y
t_LastName	Penne
t_nFirstName	Patti
t_nMiddleName	P
t_nLastName	Prosciutto

Tab-Delimited Values Shown on Right

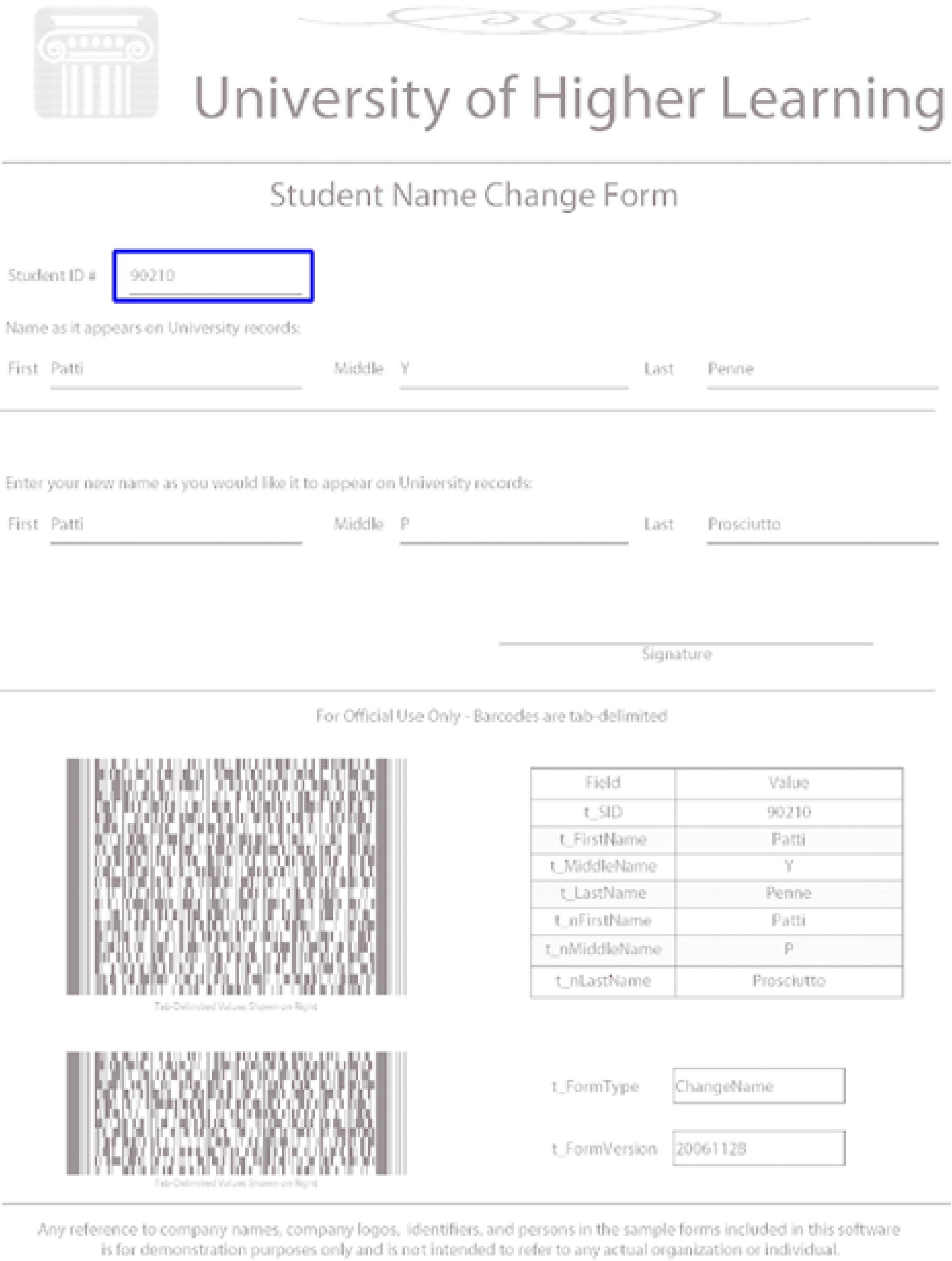
Tab-Delimited Values Shown on Right

t_FormType	ChangeName
t_FormVersion	20061128

...ce to company names, company logos, identifiers, and persons in the sample forms included in this so...
...tration purposes only and is not intended to refer to any actual organization or individual.

Another common difference between scanners can be simulated by changing the gamma of the images:

```
aug = A.Compose([
    A.RandomGamma(gamma_limit=(400, 500), always_apply=True)
], bbox_params=bbox_params)
```



University of Higher Learning

Student Name Change Form

Student ID # 90210

Name as it appears on University records:

First Patti Middle Y Last Penne

Enter your new name as you would like it to appear on University records:

First Patti Middle P Last Prosciutto

Signature

For Official Use Only - Barcodes are tab-delimited

Tab-Delimited Values Shown on Right

Field	Value
t_SID	90210
t_FirstName	Patti
t_MiddleName	Y
t_LastName	Penne
t_nFirstName	Patti
t_nMiddleName	P
t_nLastName	Prosciutto

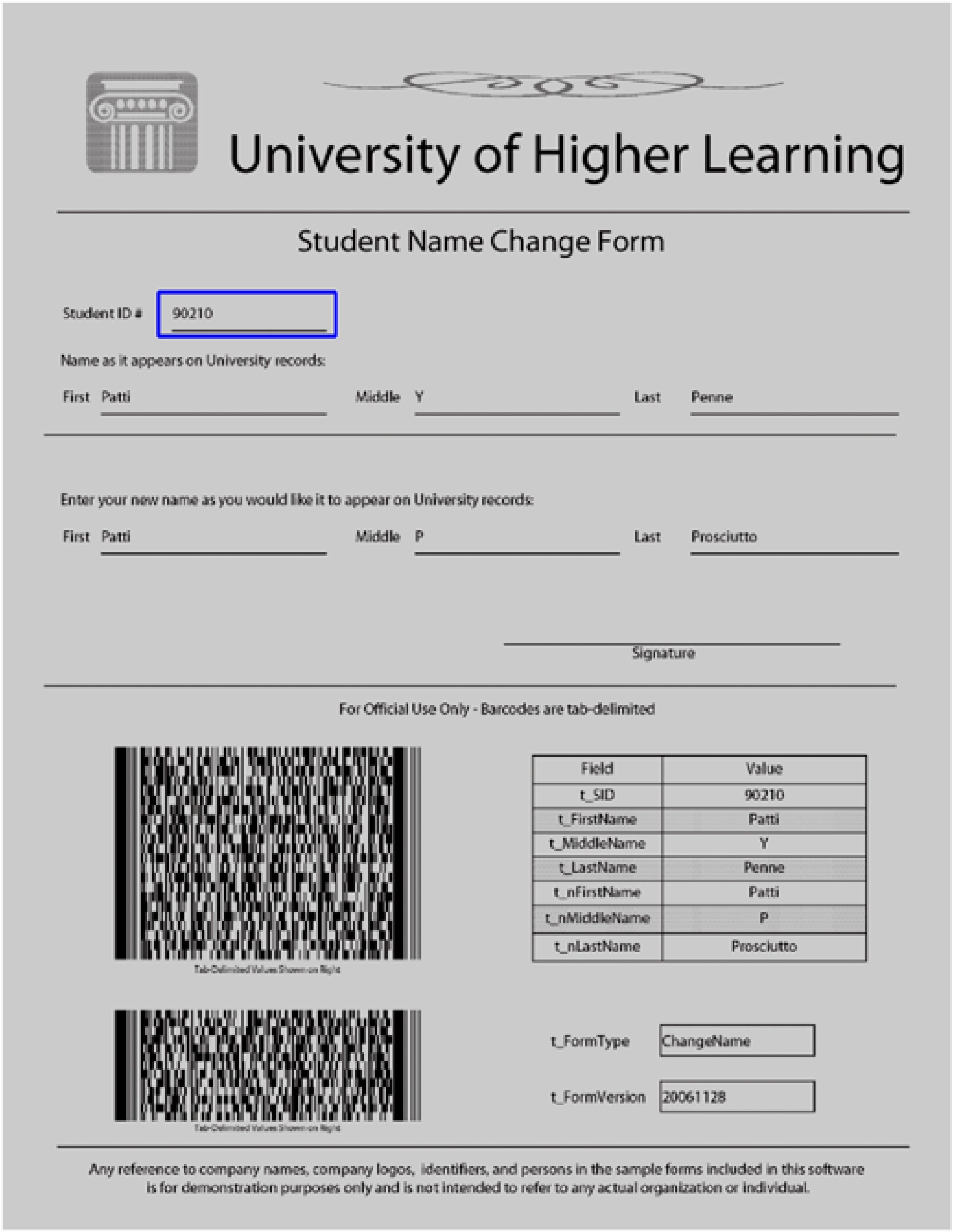
Tab-Delimited Values Shown on Right

t_FormType	ChangeName
t_FormVersion	20061128

Any reference to company names, company logos, identifiers, and persons in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization or individual.

or adjusting the brightness and contrast:

```
aug = A.Compose([
    A.RandomBrightnessContrast(always_apply=True),
], bbox_params=bbox_params)
```



University of Higher Learning

Student Name Change Form

Student ID # 90210

Name as it appears on University records:

First Patti Middle Y Last Penne

Enter your new name as you would like it to appear on University records:

First Patti Middle P Last Prosciutto

Signature

For Official Use Only - Barcodes are tab-delimited

Tab-Delimited Values Shown on Right

Field	Value
t_SID	90210
t_FirstName	Patti
t_MiddleName	Y
t_LastName	Penne
t_nFirstName	Patti
t_nMiddleName	P
t_nLastName	Prosciutto

Tab-Delimited Values Shown on Right

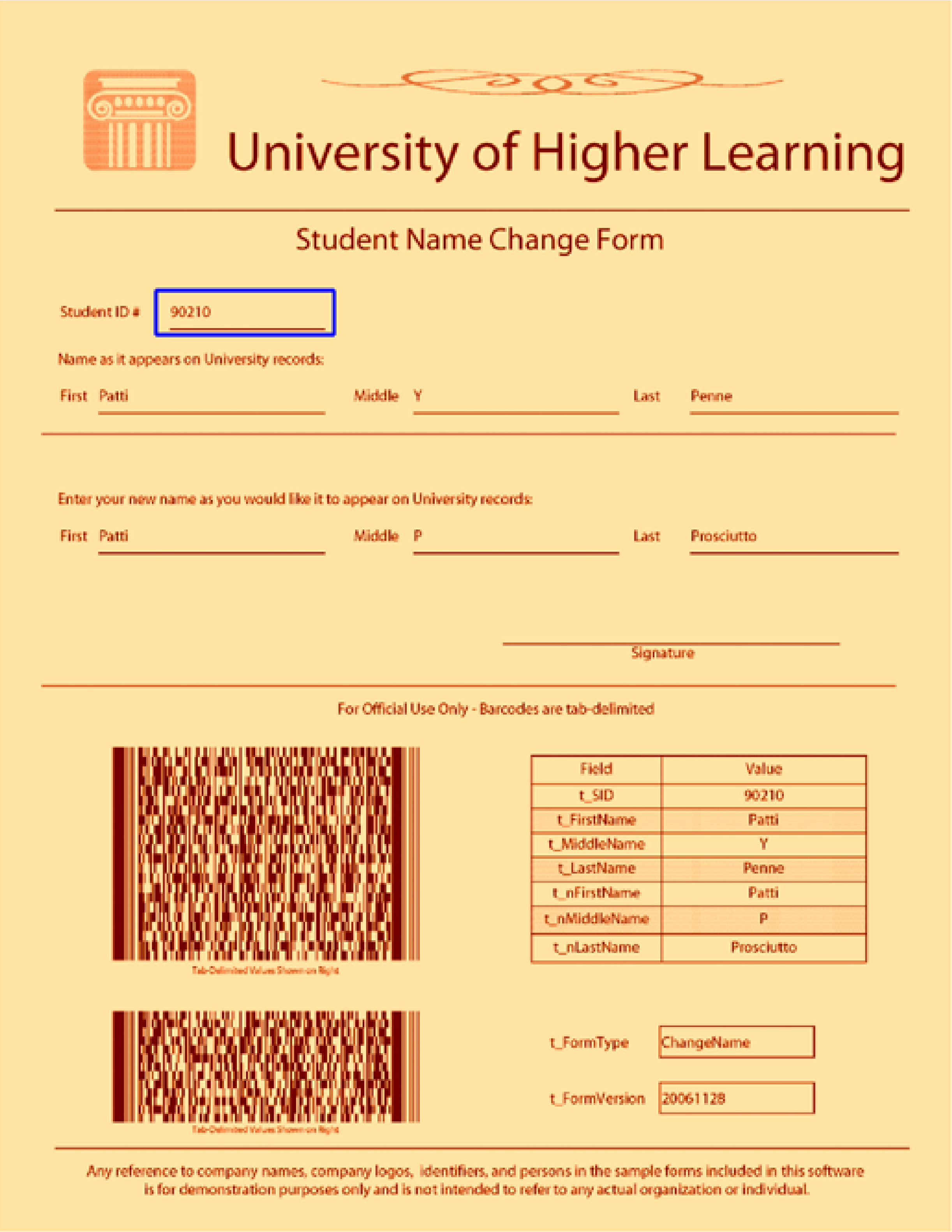
t_FormType ChangeName

t_FormVersion 20061128

Any reference to company names, company logos, identifiers, and persons in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization or individual.

Incorrect color profiles can also be simulated with `RGBShift`:

```
aug = A.Compose([
    A.RGBShift(
      always_apply=True,
      r_shift_limit=100,
      g_shift_limit=100,
      b_shift_limit=100
    ),
], bbox_params=bbox_params)
```



University of Higher Learning

Student Name Change Form

Student ID # 90210

Name as it appears on University records:

First Patti Middle Y Last Penne

Enter your new name as you would like it to appear on University records:

First Patti Middle P Last Prosciutto

Signature

For Official Use Only - Barcodes are tab-delimited

Tab-Delimited Values Shown on Right

Field	Value
t_SID	90210
t_FirstName	Patti
t_MiddleName	Y
t_LastName	Penne
t_nFirstName	Patti
t_nMiddleName	P
t_nLastName	Prosciutto

Tab-Delimited Values Shown on Right

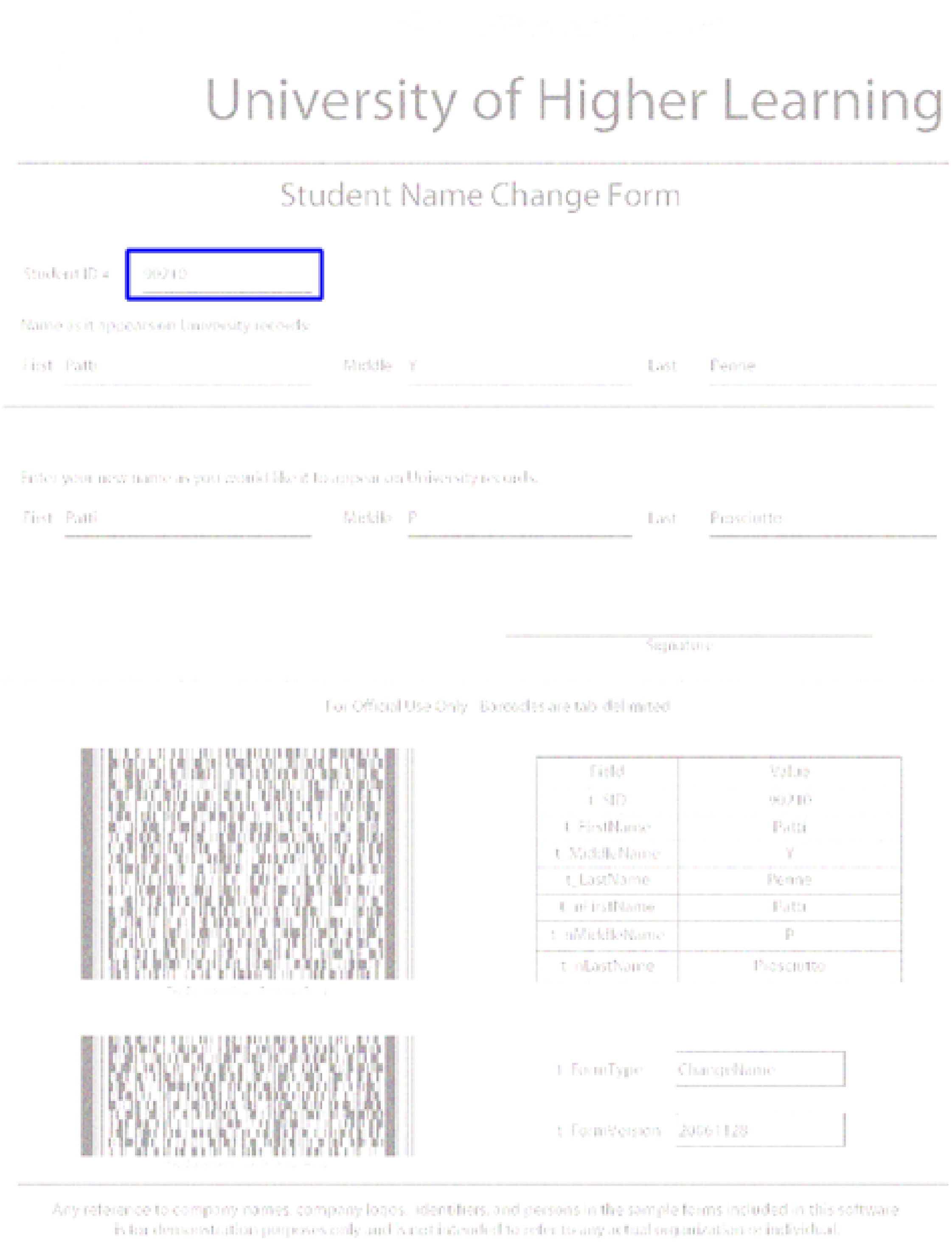
t_FormType ChangeName

t_FormVersion 20061128

Any reference to company names, company logos, identifiers, and persons in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization or individual.

You can simulate hard to read documents by applying some noise:

```
aug = A.Compose([
    A.GaussNoise(
      always_apply=True,
      var_limit=(100, 300),
      mean=150
    ),
], bbox_params=bbox_params)
```



University of Higher Learning

Student Name Change Form

Student ID # 90210

Name as it appears on University records:

First Patti Middle Y Last Penne

Enter your new name as you would like it to appear on University records.

First Patti Middle P Last Prosciutto

Signature

For Official Use Only Barcodes are tab delimited

Field	Value
t_SID	90210
t_FirstName	Patti
t_MiddleName	Y
t_LastName	Penne
t_nFirstName	Patti
t_nMiddleName	P
t_nLastName	Prosciutto

t_FormType ChangeName

t_FormVersion 20061128

Any reference to company names, company logos, identifiers, and persons in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization or individual.

Creating Augmented Dataset

You've probably guessed that you can compose multiple augmentations. You can also choose how likely is to apply the specific transformation like so:

```
doc_aug = A.Compose([
    A.Flip(p=0.25),
    A.RandomGamma(gamma_limit=(20, 300), p=0.5),
    A.RandomBrightnessContrast(p=0.85),
    A.Rotate(limit=35, p=0.9),
    A.RandomRotate90(p=0.25),
    A.RGBShift(p=0.75),
    A.GaussNoise(p=0.25)
], bbox_params=bbox_params)
```

You might want to quit with your image augmentation attempts right here. How can you correctly choose so many parameters? Furthermore, the parameters and augmentations might be highly domain-specific.

Luckily, the Albumentations Exploration Tool[291] might help you explore different parameter configurations visually. You might even try to "learn" good augmentations. Learning Data Augmentation Strategies for Object Detection[292] might be a first good read on the topic (source code included).

Object detection tasks have somewhat standard annotation format:

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

Let's create *100* augmented images and save an annotation file for those:

```
DATASET_PATH = 'data/augmented'
IMAGES_PATH = f'{DATASET_PATH}/images'

os.makedirs(DATASET_PATH, exist_ok=True)
os.makedirs(IMAGES_PATH, exist_ok=True)

rows = []
for i in tqdm(range(100)):
  augmented = doc_aug(
    image=form,
    bboxes=[STUDENT_ID_BBOX],
    field_id=['1']
  )
```

[291] https://albumentations-demo.herokuapp.com/
[292] https://arxiv.org/pdf/1906.11172v1.pdf

```
  file_name = f'form_aug_{i}.jpg'
  for bbox in augmented['bboxes']:
    x_min, y_min, x_max, y_max = map(lambda v: int(v), bbox)
    rows.append({
      'file_name': f'images/{file_name}',
      'x_min': x_min,
      'y_min': y_min,
      'x_max': x_max,
      'y_max': y_max,
      'class': 'student_id'
    })

  cv2.imwrite(f'{IMAGES_PATH}/{file_name}', augmented['image'])

pd.DataFrame(rows).to_csv(
  f'{DATASET_PATH}/annotations.csv',
  header=True,
  index=None
)
```

Note that the code is somewhat generic and can handle multiple bounding boxes per image. You should easily be able to expand this code to handle multiple images from your dataset.

Conclusion

Great job! You can now add more training data for your models by augmenting images. We just scratched the surface of the Albumentation library. Feel free to explore and build even more powerful image augmentation pipelines!

You now know how to:

- Load images using OpenCV
- Apply various image augmentations
- Compose complex augmentations to simulate real-world data
- Create augmented dataset ready to use for Object Detection

Run the complete notebook in your browser[293]

The complete project on GitHub[294]

[293] https://colab.research.google.com/drive/12r6e0grdtssEjxYAMSAnwJj3y7fVfn-V
[294] https://github.com/curiousily/Deep-Learning-For-Hackers

References

- Albumentations[295]
- A survey on Image Data Augmentation for Deep Learning[296]
- A Survey on Face Data Augmentation[297]
- Learning Data Augmentation Strategies for Object Detection[298]
- Albumentations Exploration Tool[299]

[295] https://github.com/albumentations-team/albumentations
[296] https://link.springer.com/article/10.1186/s40537-019-0197-0
[297] https://arxiv.org/pdf/1904.11685.pdf
[298] https://arxiv.org/pdf/1906.11172v1.pdf
[299] https://albumentations-demo.herokuapp.com/

Sentiment Analysis

> TL;DR Learn how to preprocess text data using the Universal Sentence Encoder model. Build a model for sentiment analysis of hotel reviews.

This tutorial will show you how to develop a Deep Neural Network for text classification (sentiment analysis). We'll skip most of the preprocessing using a pre-trained model that converts text into numeric vectors.

You'll learn how to:

- Convert text to embedding vectors using the Universal Sentence Encoder model
- Build a hotel review Sentiment Analysis model
- Use the model to predict sentiment on unseen data

Run the complete notebook in your browser[300]

The complete project on GitHub[301]

Universal Sentence Encoder

Unfortunately, Neural Networks don't understand text data. To deal with the issue, you must figure out a way to convert text into numbers. There are a variety of ways to solve the problem, but most well-performing models use Embeddings[302].

In the past, you had to do a lot of preprocessing - tokenization, stemming, remove punctuation, remove stop words, and more. Nowadays, pre-trained models offer built-in preprocessing. You might still go the manual route, but you can get a quick and dirty prototype with high accuracy by using libraries.

The Universal Sentence Encoder (USE)[303] encodes sentences into embedding vectors. The model is freely available at TF Hub[304]. It has great accuracy and supports multiple languages. Let's have a look at how we can load the model:

[300] https://colab.research.google.com/drive/1vFocnjzESxe7Mpx6NC65O28mkuuxxYI4
[301] https://github.com/curiousily/Deep-Learning-For-Hackers
[302] https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture
[303] https://arxiv.org/abs/1803.11175
[304] https://tfhub.dev/google/universal-sentence-encoder-multilingual-large/3

```
import tensorflow_hub as hub

use = hub.load("https://tfhub.dev/google/universal-sentence-encoder-multilingual-lar\
ge/3")
```

Next, let's define two sentences that have a similar meaning:

```
sent_1 = ["the location is great"]
sent_2 = ["amazing location"]
```

Using the model is really simple:

```
emb_1 = use(sent_1)
emb_2 = use(sent_2)
```

What is the result?

```
print(emb_1.shape)
```

```
TensorShape([1, 512])
```

Each sentence you pass to the model is encoded as a vector with *512* elements. You can think of *USE* as a tool to compress any textual data into a vector of fixed size while preserving the similarity between sentences.

How can we calculate the similarity between two embeddings? We can use the inner product (the values are normalized):

```
print(np.inner(emb_1, emb_2).flatten()[0])
```

```
0.79254687
```

Values closer to *1* indicate more similarity. So, those two are quite similar, indeed!

We'll use the model for the pre-processing step. Note that you can use it for almost every NLP task out there, as long as the language you're using is supported.

Hotel Reviews Data

The dataset is hosted on Kaggle[305] and is provided by Jiashen Liu[306]. It contains European hotel reviews that were scraped from Booking.com[307].

> This dataset contains 515,000 customer reviews and scoring of 1493 luxury hotels across Europe. Meanwhile, the geographical location of hotels are also provided for further analysis.

Let's load the data:

```
df = pd.read_csv("Hotel_Reviews.csv", parse_dates=['Review_Date'])
```

While the dataset is quite rich, we're interested in the review text and review score. Let's get those:

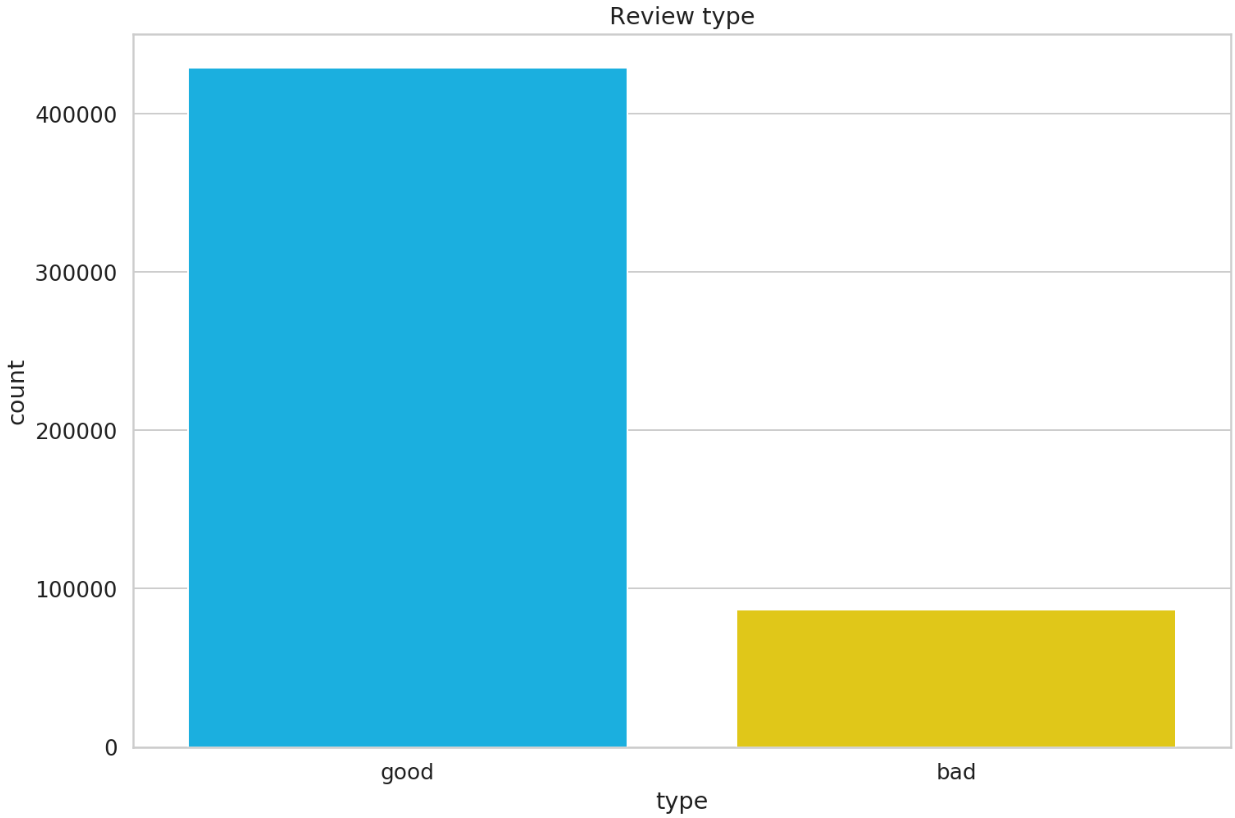
```
df["review"] = df["Negative_Review"] + df["Positive_Review"]
df["review_type"] = df["Reviewer_Score"].apply(
  lambda x: "bad" if x < 7 else "good"
)

df = df[["review", "review_type"]]
```

Any review with a score of 6 or below is marked as "bad".

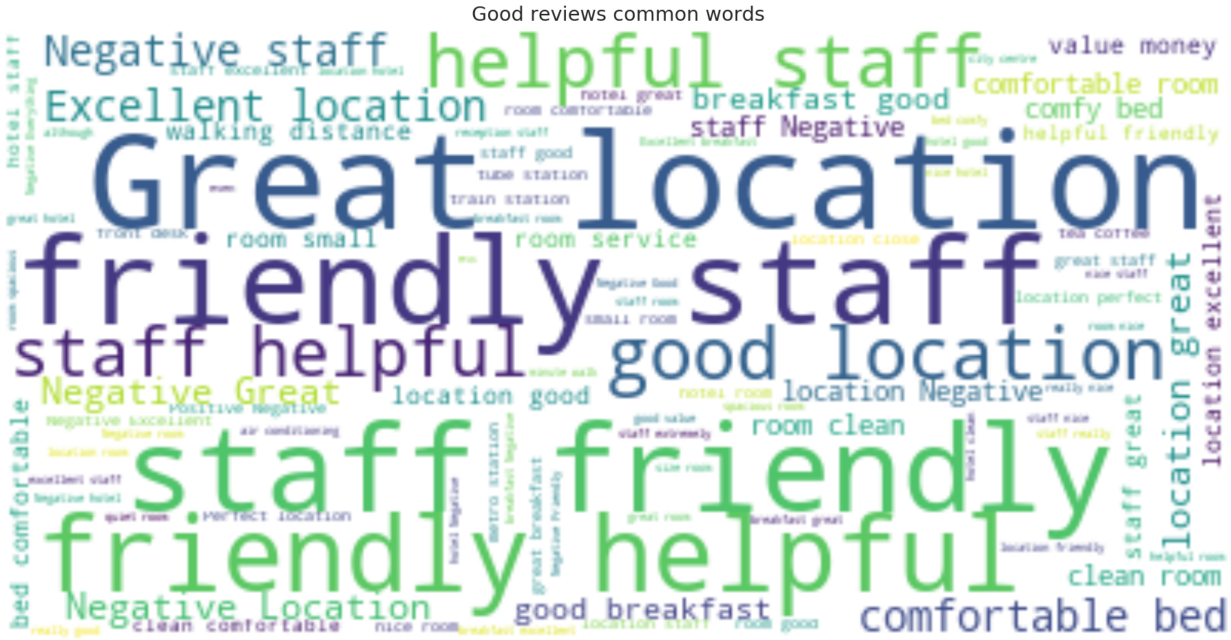
Exploration

How many of each review type we have?

[305] https://www.kaggle.com/jiashenliu/515k-hotel-reviews-data-in-europe
[306] https://www.linkedin.com/in/jiashen-liu/
[307] https://www.booking.com/

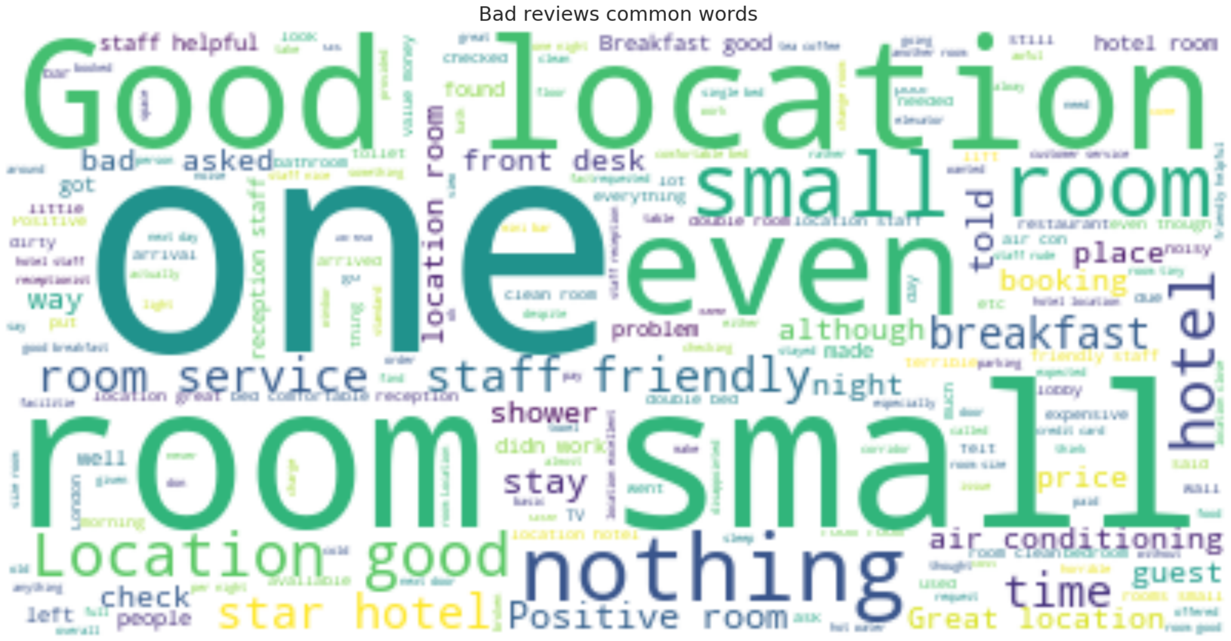


We have a severe imbalance in favor of good reviews. We'll have to do something about that. However, let's have a look at the most common words contained within the positive reviews:



“Location, location, location” - pretty common saying in the tourism business. Staff friendliness seems like the second most common quality that is important for positive reviewers.

How about the bad reviews?



Much more diverse set of phrases. Note that “good location” is still present. Room qualities are important, too!

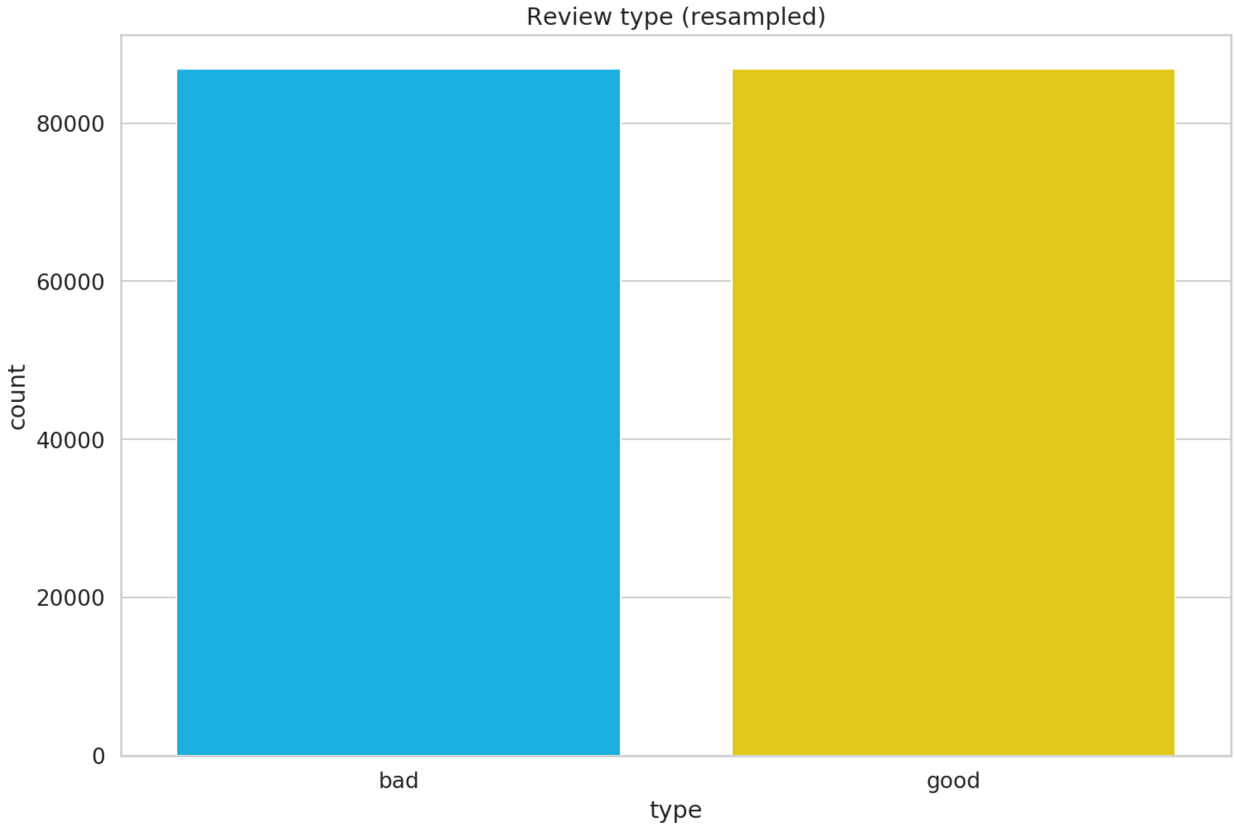
Preprocessing

We'll deal with the review type imbalance by equating the number of good ones to that of the bad ones:

```
good_df = good_reviews.sample(n=len(bad_reviews), random_state=RANDOM_SEED)
bad_df = bad_reviews
review_df = good_df.append(bad_df).reset_index(drop=True)
print(review_df.shape)
```

```
(173702, 2)
```

Let's have a look at the new review type distribution:



We have over *80k* examples for each type. Next, let's one-hot encode the review types:

```
from sklearn.preprocessing import OneHotEncoder

type_one_hot = OneHotEncoder(sparse=False).fit_transform(
  review_df.review_type.to_numpy().reshape(-1, 1)
)
```

We'll split the data for training and test datasets:

```
train_reviews, test_reviews, y_train, y_test =\
  train_test_split(
    review_df.review,
    type_one_hot,
    test_size=.1,
    random_state=RANDOM_SEED
  )
```

Finally, we can convert the reviews to embedding vectors:

```
X_train = []
for r in tqdm(train_reviews):
  emb = use(r)
  review_emb = tf.reshape(emb, [-1]).numpy()
  X_train.append(review_emb)

X_train = np.array(X_train)
```

```
X_test = []
for r in tqdm(test_reviews):
  emb = use(r)
  review_emb = tf.reshape(emb, [-1]).numpy()
  X_test.append(review_emb)

X_test = np.array(X_test)
```

```
print(X_train.shape, y_train.shape)
```

```
(156331, 512) (156331, 2)
```

We have ~*156k* training examples and somewhat equal distribution of review types. How good can we predict review sentiment with that data?

Sentiment Analysis

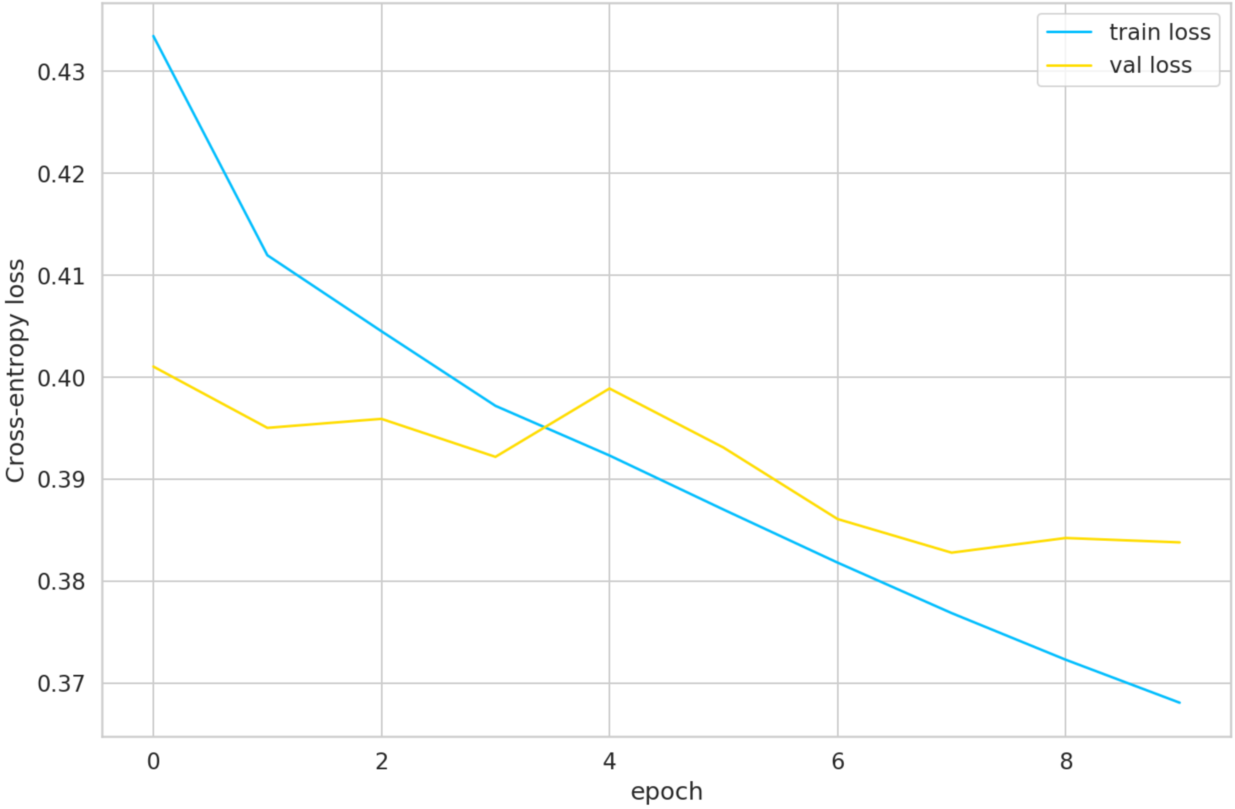
Sentiment Analysis is a binary classification problem. Let’s use Keras to build a model:

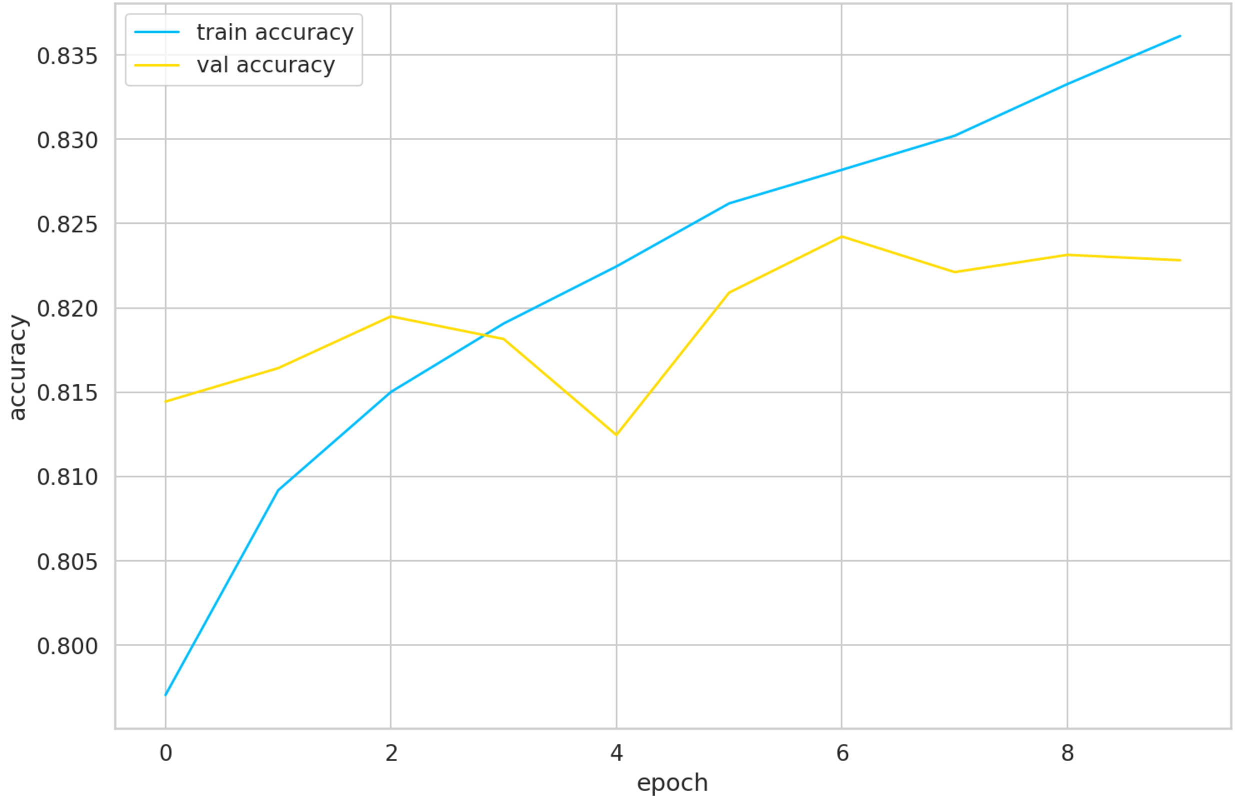
```
model = keras.Sequential()

model.add(
  keras.layers.Dense(
    units=256,
    input_shape=(X_train.shape[1], ),
    activation='relu'
  )
)
model.add(
  keras.layers.Dropout(rate=0.5)
)

model.add(
  keras.layers.Dense(
    units=128,
    activation='relu'
  )
)
model.add(
  keras.layers.Dropout(rate=0.5)
)

model.add(keras.layers.Dense(2, activation='softmax'))
model.compile(
    loss='categorical_crossentropy',
    optimizer=keras.optimizers.Adam(0.001),
    metrics=['accuracy']
)
```

The model is composed of 2 fully-connected hidden layers. Dropout is used for regularization.

We’ll train for 10 epochs and use 10% of the data for validation:

```
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=16,
    validation_split=0.1,
    verbose=1,
    shuffle=True
)
```





Our model is starting to overfit at about epoch 8, so we'll not train for much longer. We got about 82% accuracy on the validation set. Let's evaluate on the test set:

```
model.evaluate(X_test, y_test)
```

```
[0.39665538506298975, 0.82044786]
```

82% accuracy on the test set, too!

Predicting Sentiment

Let's make some predictions:

```
print(test_reviews.iloc[0])
print("Bad" if y_test[0][0] == 1 else "Good")
```

> Asked for late checkout and didnt get an answer then got a yes but had to pay 25 euros by noon they called to say sorry you have to leave in 1h knowing that i had a sick dog and an appointment next to the hotel Location staff

> Bad

The prediction:

```
y_pred = model.predict(X_test[:1])
print(y_pred)
"Bad" if np.argmax(y_pred) == 0 else "Good"
```

```
[[0.9274073  0.07259267]]
'Bad'
```

This one is correct, let's have a look at another one:

```
print(test_reviews.iloc[1])
print("Bad" if y_test[1][0] == 1 else "Good")
```

> Don t really like modern hotels Had no character Bed was too hard Good location rooftop pool new hotel nice balcony nice breakfast
>
> Good

```
y_pred = model.predict(X_test[1:2])
print(y_pred)
"Bad" if np.argmax(y_pred) == 0 else "Good"
```

```
[[0.39992586 0.6000741 ]]
'Good'
```

Conclusion

Well done! You can now build a Sentiment Analysis model with Keras. You can reuse the model and do any text classification task, too!

You learned how to:

- Convert text to embedding vectors using the Universal Sentence Encoder model
- Build a hotel review Sentiment Analysis model
- Use the model to predict sentiment on unseen data

Run the complete notebook in your browser[308]

The complete project on GitHub[309]

Can you use the Universal Sentence Encoder model for other tasks? Comment down below.

[308] https://colab.research.google.com/drive/1vFocnjzESxe7Mpx6NC65O28mkuuxxYI4

[309] https://github.com/curiousily/Deep-Learning-For-Hackers

References

- Universal Sentence Encoder[310]
- Word embeddings[311]
- 515k hotel reviews on Kaggle[312]

[310] https://arxiv.org/abs/1803.11175
[311] https://www.tensorflow.org/tutorials/text/word_embeddings
[312] https://www.kaggle.com/jiashenliu/515k-hotel-reviews-data-in-europe

Intent Recognition with BERT

> TL;DR Learn how to fine-tune the BERT model for text classification. Train and evaluate it on a small dataset for detecting seven intents. The results might surprise you!

Recognizing intent (IR) from text is very useful these days. Usually, you get a short text (sentence or two) and have to classify it into one (or multiple) categories.

Multiple product support systems (help centers) use IR to reduce the need for a large number of employees that copy-and-paste boring responses to frequently asked questions. Chatbots, automated email responders, answer recommenders (from a knowledge base with questions and answers) strive to not let you take the time of a real person.

This guide will show you how to use a pre-trained NLP model that might solve the (technical) support problem that many business owners have. I mean, BERT is freaky good! It is really easy to use, too!

Run the complete notebook in your browser[313]

The complete project on GitHub[314]

Data

The data contains various user queries categorized into seven intents. It is hosted on GitHub[315] and is first presented in this paper[316].

Here are the intents:

- SearchCreativeWork (e.g. Find me the I, Robot television show)
- GetWeather (e.g. Is it windy in Boston, MA right now?)
- BookRestaurant (e.g. I want to book a highly rated restaurant for me and my boyfriend tomorrow night)
- PlayMusic (e.g. Play the last track from Beyoncé off Spotify)
- AddToPlaylist (e.g. Add Diamonds to my roadtrip playlist)
- RateBook (e.g. Give 6 stars to Of Mice and Men)
- SearchScreeningEvent (e.g. Check the showtimes for Wonder Woman in Paris)

I've done a bit of preprocessing and converted the JSON files into easy to use/load CSVs. Let's download them:

[313] https://colab.research.google.com/drive/1WQY_XxdiCVFzjMXnDdNfUjDFi0CN5hkT

[314] https://github.com/curiousily/Deep-Learning-For-Hackers

[315] https://github.com/snipsco/nlu-benchmark/tree/master/2017-06-custom-intent-engines

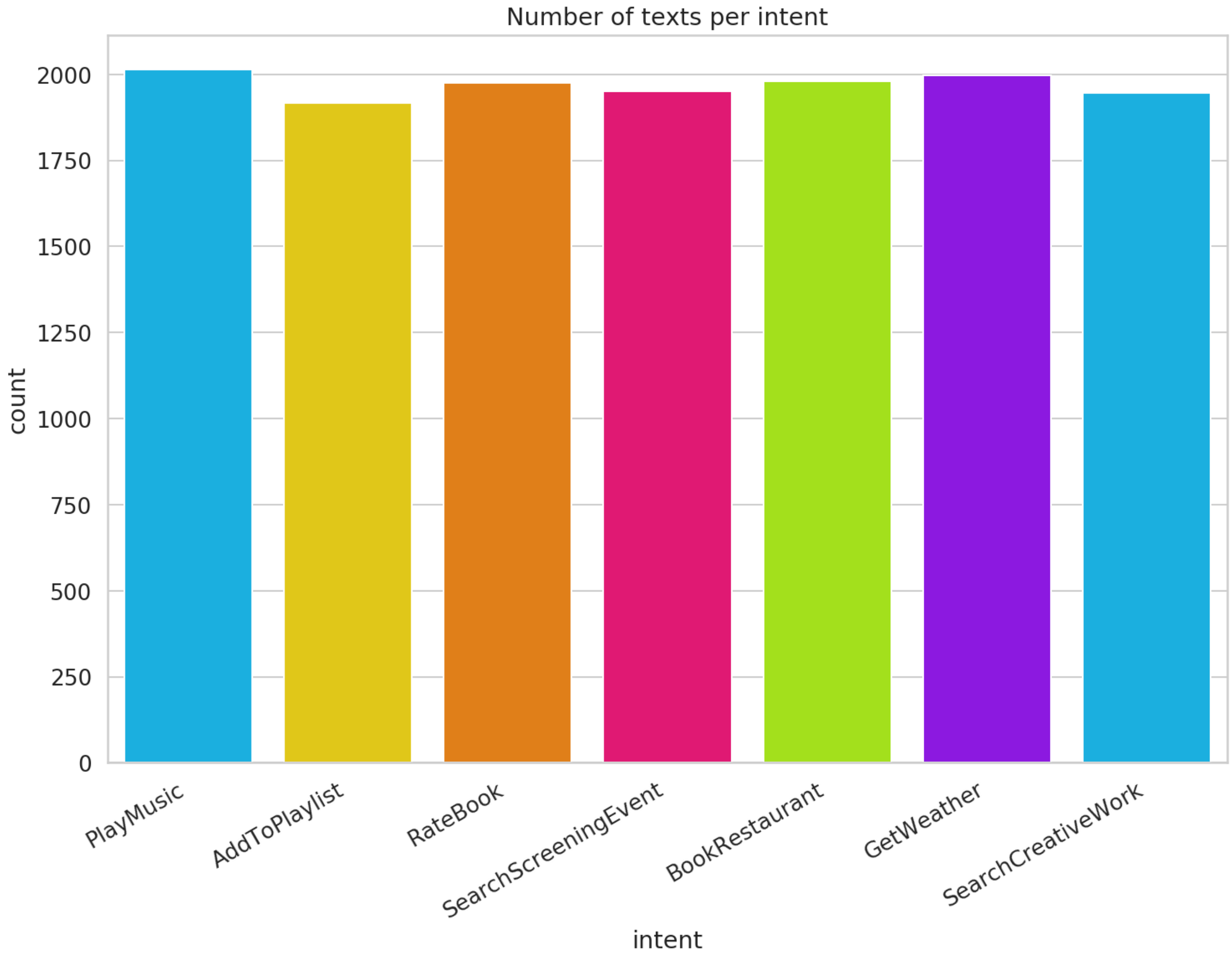
[316] https://arxiv.org/abs/1805.10190

```
!gdown --id 1OlcvGWReJMuyYQuOZm149vHWwPtlboR6 --output train.csv
!gdown --id 1Oi5cRlTybuIF2Fl5Bfsr-KkqrXrdt77w --output valid.csv
!gdown --id 1ep9H6-HvhB4utJRLVcLzieWNUSG3P_uF --output test.csv
```

We'll load the data into data frames and expand the training data by merging the training and validation intents:

```
train = pd.read_csv("train.csv")
valid = pd.read_csv("valid.csv")
test = pd.read_csv("test.csv")

train = train.append(valid).reset_index(drop=True)
```

We have `13,784` training examples and two columns - `text` and `intent`. Let's have a look at the number of texts per intent:



The amount of texts per intent is quite balanced, so we'll not be needing any imbalanced modeling techniques.

BERT

The BERT (Bidirectional Encoder Representations from Transformers) model, introduced in the BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding[317] paper, made possible achieving State-of-the-art results in a variety of NLP tasks, for the regular ML practitioner. And you can do it without having a large dataset! But how is this possible?

BERT is a pre-trained Transformer Encoder stack. It is trained on Wikipedia and the Book Corpus[318] dataset. It has two versions - Base (12 encoders) and Large (24 encoders).

BERT is built on top of multiple clever ideas by the NLP community. Some examples are ELMo[319], The Transformer[320], and the OpenAI Transformer[321].

ELMo introduced contextual word embeddings (one word can have a different meaning based on the words around it). The Transformer uses attention mechanisms to understand the context in which the word is being used. That context is then encoded into a vector representation. In practice, it does a better job with long-term dependencies.

BERT is a bidirectional model (looks both forward and backward). And the best of all, BERT can be easily used as a feature extractor or fine-tuned with small amounts of data. How good is it at recognizing intent from text?

Intent Recognition with BERT

Luckily, the authors of the BERT paper open-sourced their work[322] along with multiple pre-trained models. The original implementation is in TensorFlow, but there are very good PyTorch implementations[323] too!

Let's start by downloading one of the simpler pre-trained models and unzip it:

```
!wget https://storage.googleapis.com/bert_models/2018_10_18/uncased_L-12_H-768_A-12.\
zip
!unzip uncased_L-12_H-768_A-12.zip
```

This will unzip a checkpoint, config, and vocabulary, along with other files.

Unfortunately, the original implementation is not compatible with TensorFlow 2. The bert-for-tf2[324] package solves this issue.

[317] https://arxiv.org/abs/1810.04805
[318] https://arxiv.org/pdf/1506.06724.pdf
[319] https://arxiv.org/abs/1802.05365
[320] https://arxiv.org/abs/1706.03762
[321] https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf
[322] https://github.com/google-research/bert
[323] https://github.com/huggingface/transformers
[324] https://github.com/kpe/bert-for-tf2

Preprocessing

We need to convert the raw texts into vectors that we can feed into our model. We'll go through 3 steps:

- Tokenize the text
- Convert the sequence of tokens into numbers
- Pad the sequences so each one has the same length

Let's start by creating the BERT tokenizer:

```
tokenizer = FullTokenizer(
  vocab_file=os.path.join(bert_ckpt_dir, "vocab.txt")
)
```

Let's take it for a spin:

```
tokenizer.tokenize("I can't wait to visit Bulgaria again!")
```

```
['i', 'can', "'", 't', 'wait', 'to', 'visit', 'bulgaria', 'again', '!']
```

The tokens are in lowercase and the punctuation is available. Next, we'll convert the tokens to numbers. The tokenizer can do this too:

```
tokens = tokenizer.tokenize("I can't wait to visit Bulgaria again!")
tokenizer.convert_tokens_to_ids(tokens)
```

```
[1045, 2064, 1005, 1056, 3524, 2000, 3942, 8063, 2153, 999]
```

We'll do the padding part ourselves. You can also use the Keras padding utils for that part.

We'll package the preprocessing into a class that is heavily based on the one from this notebook[325]:

[325] https://github.com/kpe/bert-for-tf2/blob/master/examples/gpu_movie_reviews.ipynb

```
class IntentDetectionData:
  DATA_COLUMN = "text"
  LABEL_COLUMN = "intent"

  def __init__(
    self,
    train,
    test,
    tokenizer: FullTokenizer,
    classes,
    max_seq_len=192
  ):
    self.tokenizer = tokenizer
    self.max_seq_len = 0
    self.classes = classes

    train, test = map(lambda df:
      df.reindex(
        df[IntentDetectionData.DATA_COLUMN].str.len().sort_values().index
      ),
      [train, test]
    )

    ((self.train_x, self.train_y), (self.test_x, self.test_y)) =\
     map(self._prepare, [train, test])

    print("max seq_len", self.max_seq_len)
    self.max_seq_len = min(self.max_seq_len, max_seq_len)
    self.train_x, self.test_x = map(
      self._pad,
      [self.train_x, self.test_x]
    )

  def _prepare(self, df):
    x, y = [], []

    for _, row in tqdm(df.iterrows()):
      text, label =\
       row[IntentDetectionData.DATA_COLUMN], \
       row[IntentDetectionData.LABEL_COLUMN]
      tokens = self.tokenizer.tokenize(text)
      tokens = ["[CLS]"] + tokens + ["[SEP]"]
      token_ids = self.tokenizer.convert_tokens_to_ids(tokens)
      self.max_seq_len = max(self.max_seq_len, len(token_ids))
      x.append(token_ids)
      y.append(self.classes.index(label))

    return np.array(x), np.array(y)

  def _pad(self, ids):
    x = []
    for input_ids in ids:
      input_ids = input_ids[:min(len(input_ids), self.max_seq_len - 2)]
      input_ids = input_ids + [0] * (self.max_seq_len - len(input_ids))
      x.append(np.array(input_ids))
    return np.array(x)
```

We figure out the padding length by taking the minimum between the longest text and the max sequence length parameter. We also surround the tokens for each text with two special tokens: start with `[CLS]` and end with `[SEP]`.

Fine-tuning

Let's make BERT usable for text classification! We'll load the model and attach a couple of layers on it:

```
def create_model(max_seq_len, bert_ckpt_file):

  with tf.io.gfile.GFile(bert_config_file, "r") as reader:
      bc = StockBertConfig.from_json_string(reader.read())
      bert_params = map_stock_config_to_params(bc)
      bert_params.adapter_size = None
      bert = BertModelLayer.from_params(bert_params, name="bert")

  input_ids = keras.layers.Input(
    shape=(max_seq_len, ),
    dtype='int32',
    name="input_ids"
  )
  bert_output = bert(input_ids)

  print("bert shape", bert_output.shape)

  cls_out = keras.layers.Lambda(lambda seq: seq[:, 0, :])(bert_output)
  cls_out = keras.layers.Dropout(0.5)(cls_out)
  logits = keras.layers.Dense(units=768, activation="tanh")(cls_out)
  logits = keras.layers.Dropout(0.5)(logits)
  logits = keras.layers.Dense(
    units=len(classes),
    activation="softmax"
  )(logits)

  model = keras.Model(inputs=input_ids, outputs=logits)
  model.build(input_shape=(None, max_seq_len))

  load_stock_weights(bert, bert_ckpt_file)

  return model
```

We're fine-tuning the pre-trained BERT model using our inputs (text and intent). We also flatten the output and add Dropout with two Fully-Connected layers. The last layer has a softmax activation function. The number of outputs is equal to the number of intents we have - seven.

You can now use BERT to recognize intents!

Training

It is time to put everything together. We'll start by creating the data object:

```
classes = train.intent.unique().tolist()

data = IntentDetectionData(
  train,
  test,
  tokenizer,
  classes,
  max_seq_len=128
)
```

We can now create the model using the maximum sequence length:

```
model = create_model(data.max_seq_len, bert_ckpt_file)
```

Looking at the model summary:

```
model.summar()
```

You'll notice that even this "slim" BERT has almost 110 million parameters. Indeed, your model is HUGE (that's what she said).

Fine-tuning models like BERT is both art and doing tons of failed experiments. Fortunately, the authors made some recommendations:

- Batch size: 16, 32
- Learning rate (Adam): 5e-5, 3e-5, 2e-5
- Number of epochs: 2, 3, 4

```
model.compile(
  optimizer=keras.optimizers.Adam(1e-5),
  loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")]
)
```

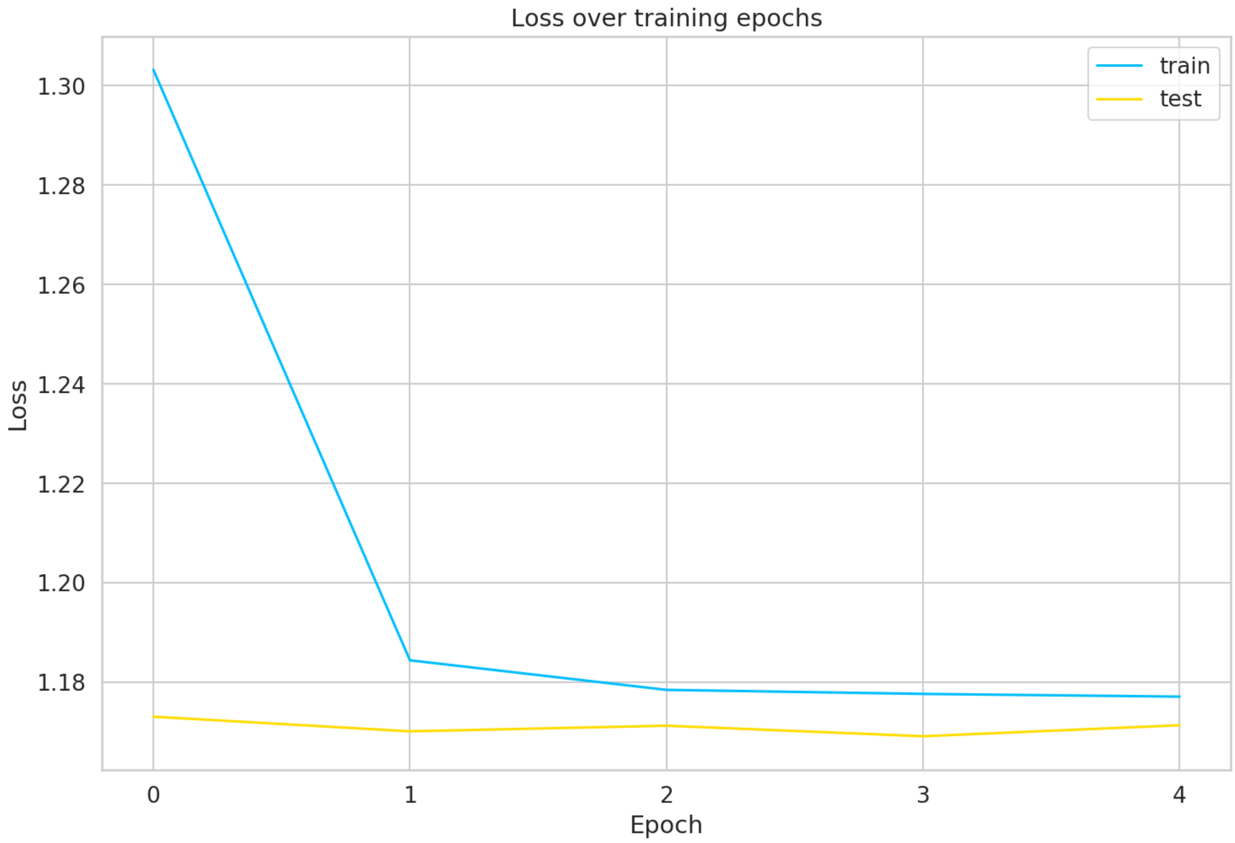
We'll use Adam with a slightly different learning rate (cause we're badasses) and use sparse categorical crossentropy, so we don't have to one-hot encode our labels.

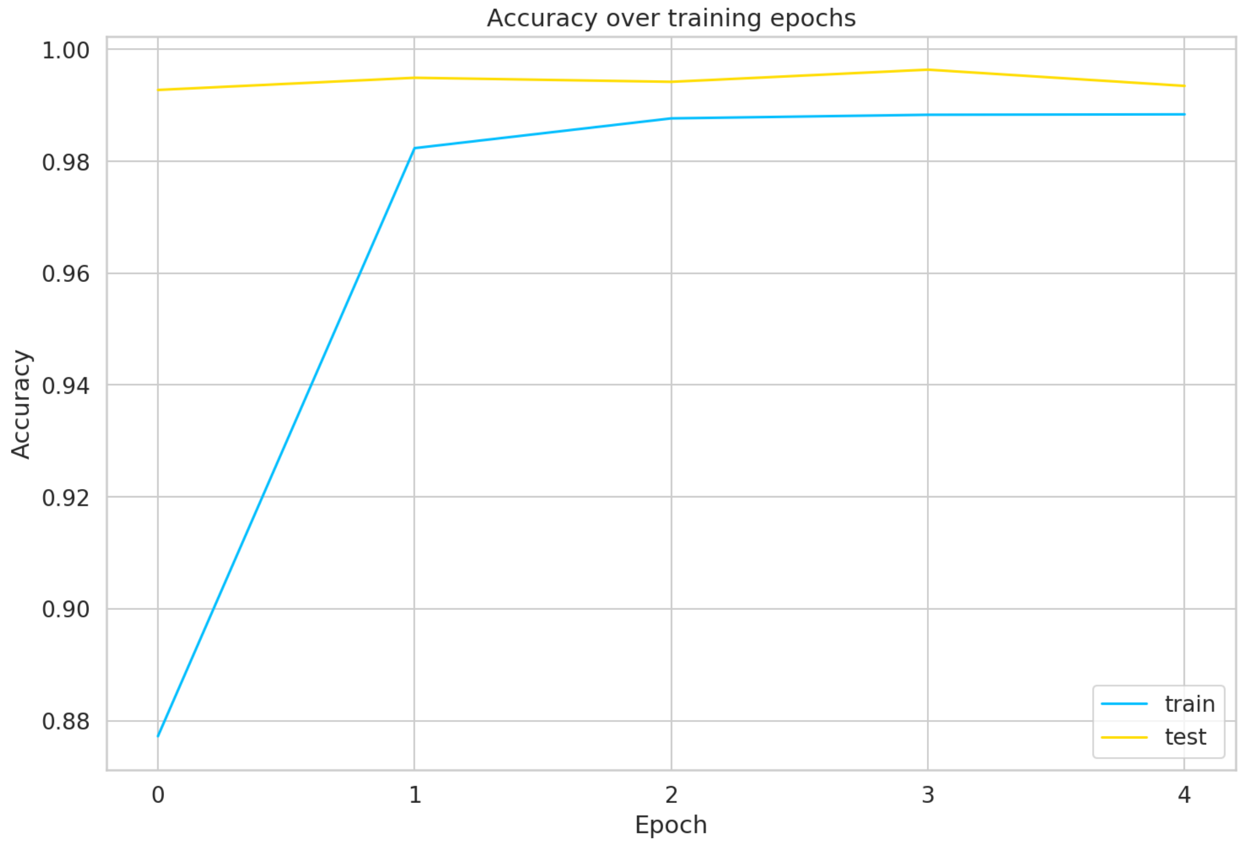
Let's fit the model:

```
log_dir = "log/intent_detection/" +\
 datetime.datetime.now().strftime("%Y%m%d-%H%M%s")
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=log_dir)

model.fit(
  x=data.train_x,
  y=data.train_y,
  validation_split=0.1,
  batch_size=16,
  shuffle=True,
  epochs=5,
  callbacks=[tensorboard_callback]
)
```

We store the training logs, so you can explore the training process in Tensorboard[326]. Let's have a look:

[326] https://www.tensorflow.org/tensorboard


Loss over training epochs
train
test
1.30
1.28
1.26
1.24
1.22
1.20
1.18
Loss
0
1
2
3
4
Epoch



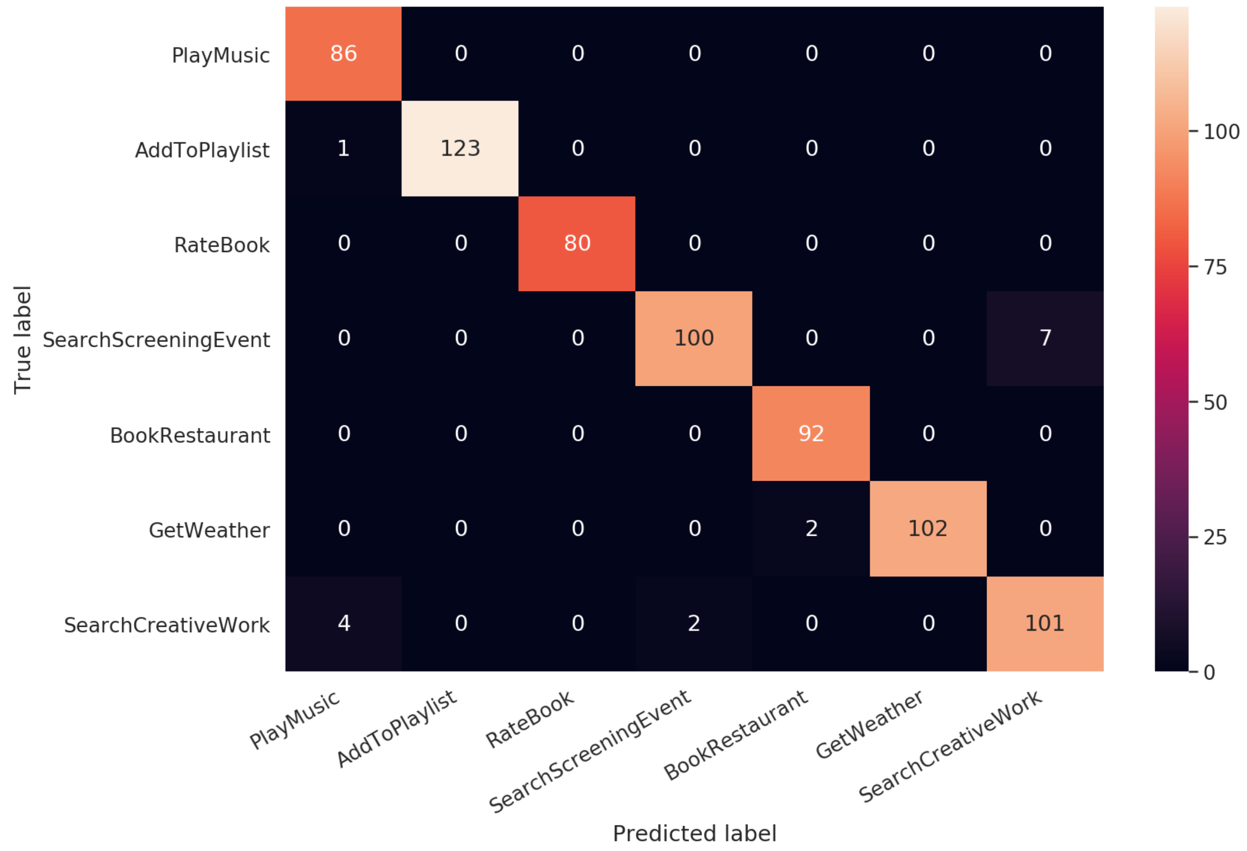
Evaluation

I got to be honest with you. I was impressed with the results. Training using only 12.5k samples we got:

```
_, train_acc = model.evaluate(data.train_x, data.train_y)
_, test_acc = model.evaluate(data.test_x, data.test_y)

print("train acc", train_acc)
print("test acc", test_acc)
```

```
train acc 0.9915119
test acc 0.9771429
```

Impressive, right? Let's have a look at the confusion matrix:



Finally, let's use the model to detect intent from some custom sentences:

```
sentences = [
  "Play our song now",
  "Rate this book as awful"
]

pred_tokens = map(tokenizer.tokenize, sentences)
pred_tokens = map(lambda tok: ["[CLS]"] + tok + ["[SEP]"], pred_tokens)
pred_token_ids = list(map(tokenizer.convert_tokens_to_ids, pred_tokens))

pred_token_ids = map(
  lambda tids: tids +[0]*(data.max_seq_len-len(tids)),
  pred_token_ids
)
pred_token_ids = np.array(list(pred_token_ids))

predictions = model.predict(pred_token_ids).argmax(axis=-1)

for text, label in zip(sentences, predictions):
  print("text:", text, "\nintent:", classes[label])
  print()
```

```
text: Play our song now
intent: PlayMusic

text: Rate this book as awful
intent: RateBook
```

Man, that's (clearly) gangsta! Ok, the examples might not be as diverse as real queries might be. But hey, go ahead and try it on your own!

Conclusion

You now know how to fine-tune a BERT model for text classification. You probably already know that you can use it for a variety of other tasks, too! You just have to fiddle with the layers. EASY!

Run the complete notebook in your browser[327]

The complete project on GitHub[328]

Doing AI/ML feels a lot like having superpowers, right? Thanks to the wonderful NLP community, you can have superpowers, too! What will you use them for?

References

- BERT Fine-Tuning Tutorial with PyTorch[329]
- SNIPS dataset[330]
- The Illustrated BERT, ELMo, and co. [331]
- BERT for dummies — Step by Step Tutorial[332]
- Multi-label Text Classification using BERT – The Mighty Transformer[333]

[327] https://colab.research.google.com/drive/1WQY_XxdiCVFzjMXnDdNfUjDFi0CN5hkT

[328] https://github.com/curiousily/Deep-Learning-For-Hackers

[329] https://mccormickml.com/2019/07/22/BERT-fine-tuning/

[330] https://github.com/snipsco/nlu-benchmark/tree/master/2017-06-custom-intent-engines

[331] https://jalammar.github.io/illustrated-bert/

[332] https://towardsdatascience.com/bert-for-dummies-step-by-step-tutorial-fb90890ffe03

[333] https://medium.com/huggingface/multi-label-text-classification-using-bert-the-mighty-transformer-69714fa3fb3d