Cycling Data explorer

This notebook outlines some of the routines available in `data_analysis.py`.

We investigate two bike rides - one a 100km outdoor ride in the Surrey Hills, the other an indoor ride on a Wattbike. We demonstrate some feature extraction methods and data visualization, and use a model relating Wattbike power to measured speed to estimate power output on the longer outdoor ride (for which power data was not available).

[1]
```
%matplotlib inline
%load_ext autoreload
%autoreload 2

import numpy as np
from bs4 import BeautifulSoup
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler,
PolynomialFeatures
from sklearn.linear_model import LinearRegression, SGDRegressor
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline
from data_analysis import *

pd.plotting.register_matplotlib_converters()
```

Papermill - Parametrized

[2]
```
FULL_RUN = True
```

File loading is easy:

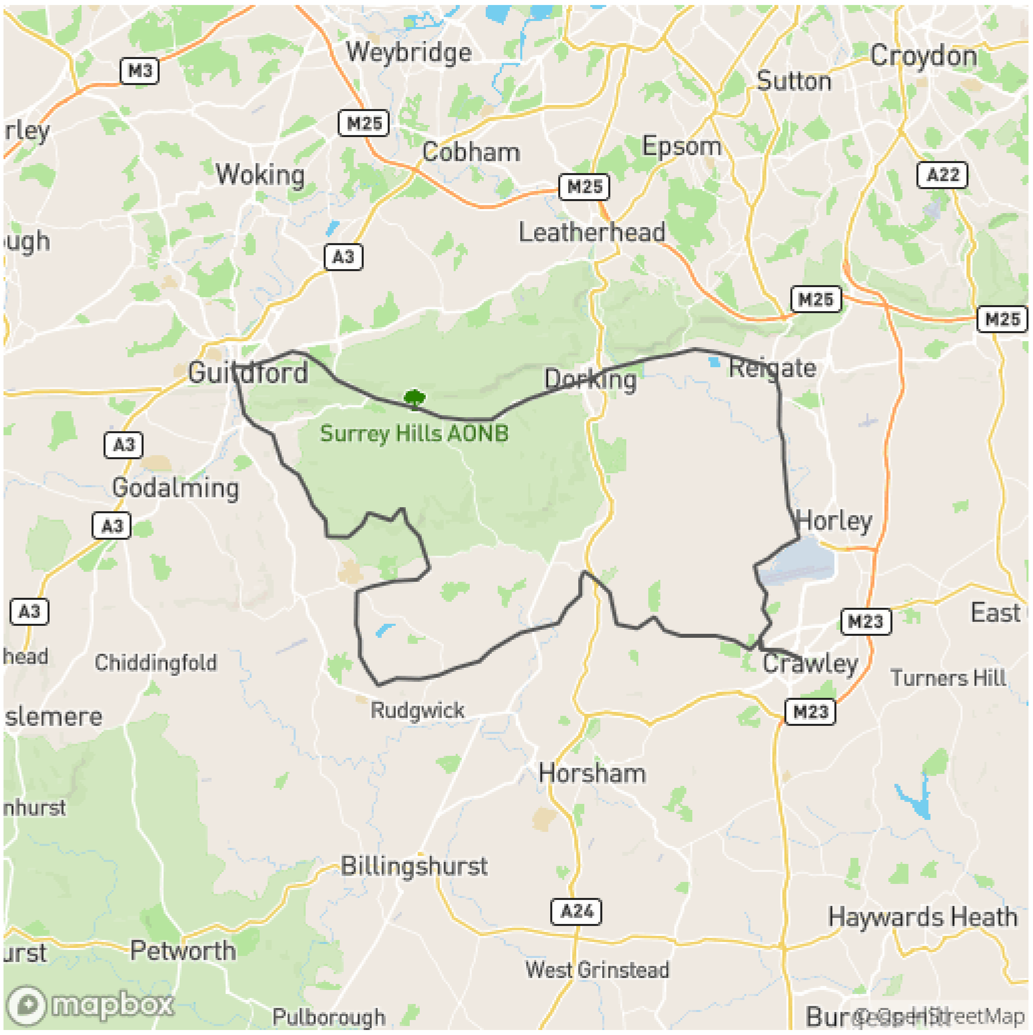
[3]
```
%time df_gpx =
load_file('Lucozade_sport_is_the_best_drink_of_all_time.gpx')
%time df_tcx = load_file('5G0G6PsD9V_jOo5zdK0S4.tcx')
df_gpx.head(3)
```

```
CPU times: user 6.05 s, sys: 191 ms, total: 6.24 s
Wall time: 6.49 s
CPU times: user 806 ms, sys: 24.7 ms, total: 831 ms
Wall time: 843 ms
```

	Cadence	Elevation	HeartRateBpm	Latitude	Longitude

Time	Cadence	Elevation	HeartRateBpm	Latitude	Longitude
Time					
2019-06-01 09:31:56	62.0	83.8	91.0	51.112387	-0.186590
2019-06-01 09:31:57	62.0	83.8	91.0	51.112385	-0.186592
2019-06-01 09:31:58	62.0	83.8	91.0	51.112383	-0.186594

And grab an image cutout of the bike ride:



We ought to augment this data with calculated values (feature extraction), for example our gpx file doesn't have a record of distance travelled. We calculate this using the haversine formula and the recorded latitude and longitude:

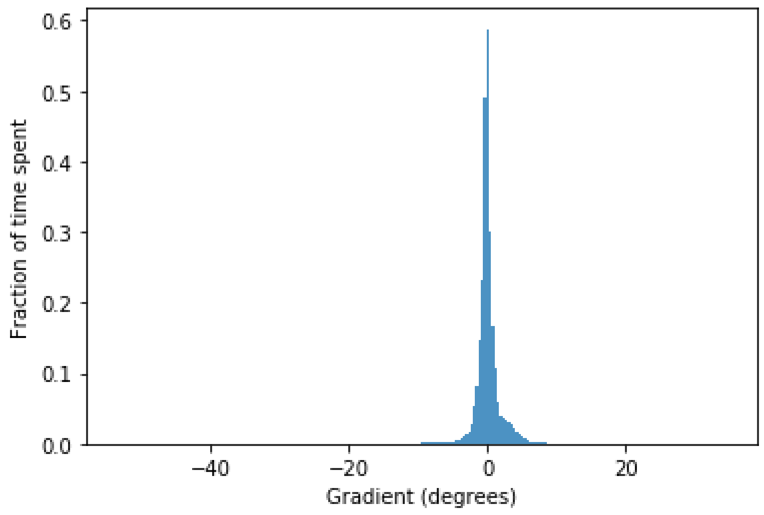
```
[5]   df_gpx['DistanceMeters'] = get_distance(df_gpx)
      df_gpx['Distance'] = df_gpx['DistanceMeters'] / 1000  # distance
      in km, useful for plotting
```

We can do the same for speed:

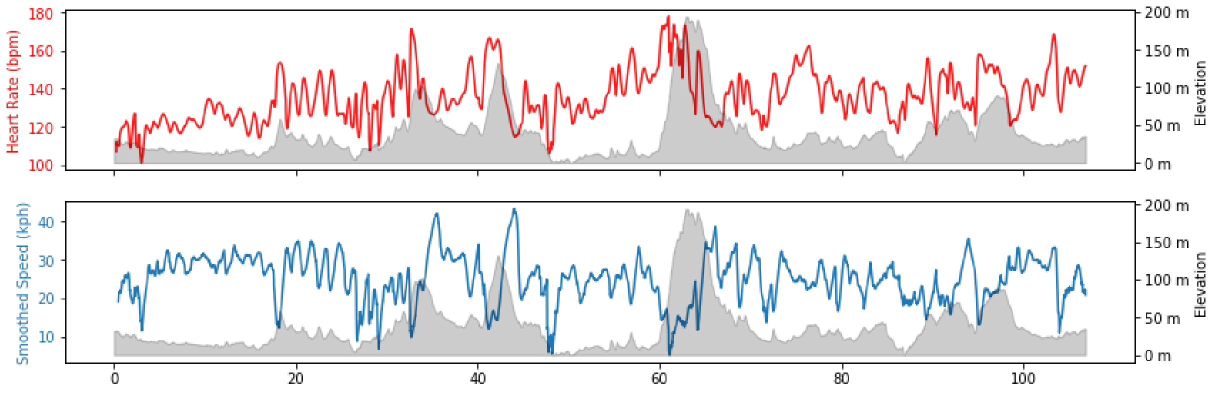
```
[6]   df_gpx['Speed'] = get_speed(df_gpx)
```

And use Elevation and Distance to calculate gradient:

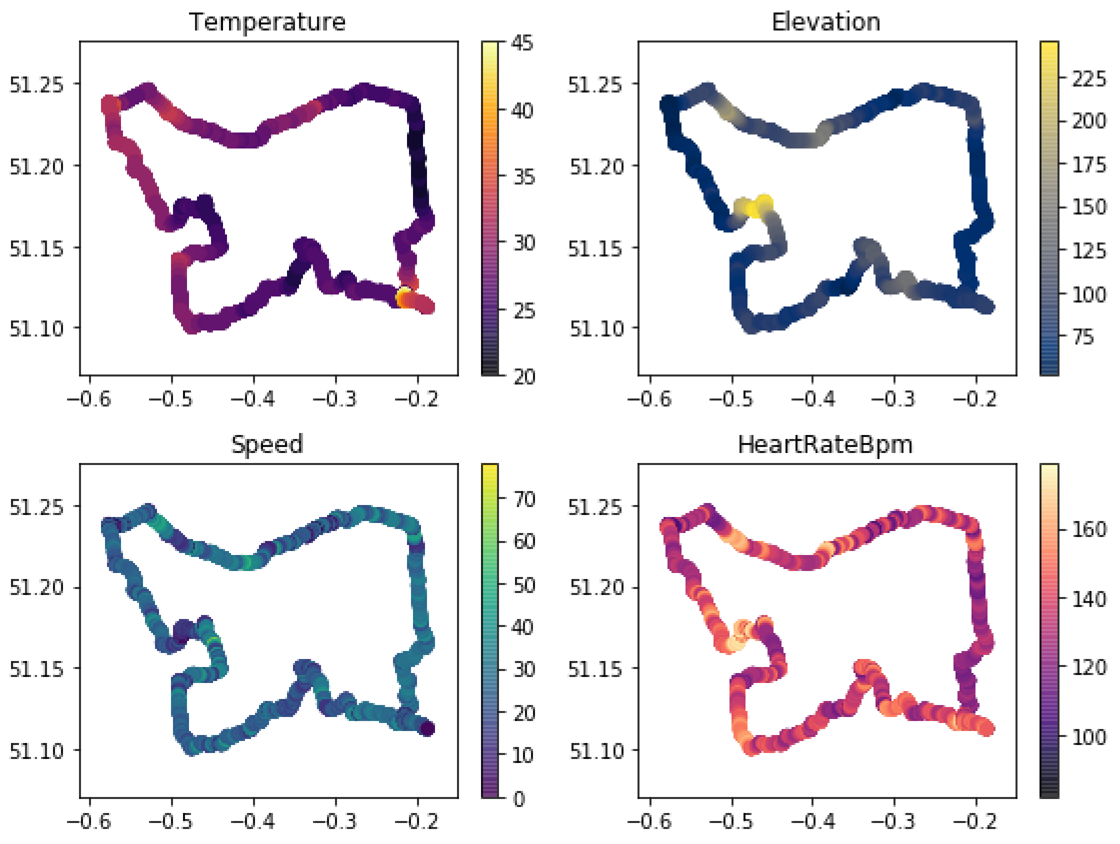
```
[7]   df_gpx['Gradient'] = get_gradient(df_gpx)
      plt.hist(df_gpx['Gradient'].dropna(), density=True, alpha=0.8,
      bins='scott')
      plt.xlabel('Gradient (degrees)')
      plt.ylabel('Fraction of time spent');
```



Now we can make those lovely plots you see on Strava (and other pain-monitoring sites)

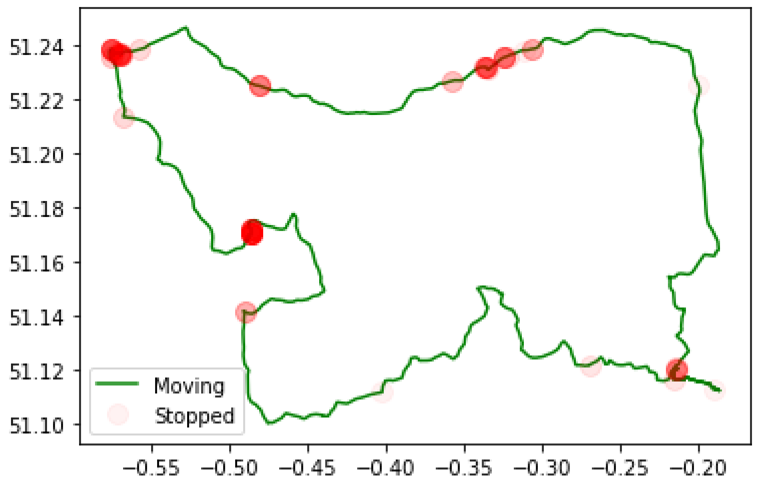


We can also spatially plot things like elevation, speed and heart rate, though not easily on a map (curse you mapbox for not allowing bounding box queries easily)



Let's see where in the ride I needed a break (or was stuck at lights or in traffic...):

```
[10]  mask = df_gpx['Distance'].diff() != 0
      plt.plot(
          df_gpx['Longitude'][mask], df_gpx['Latitude'][mask],
          'g', markersize=1, label='Moving',
      )
      plt.plot(
          df_gpx['Longitude'][~mask], df_gpx['Latitude'][~mask],
        'or', alpha=1/20, markersize=10, label='Stopped',
    )
    plt.legend();
```



Power data

Okay, we've shown it's pretty easy to interact with gpx data (location, elevation & extensions), but the big bit of data missing here is **Power!**

We'll load a file from an interval session on a Wattbike, and see what we can learn from it!

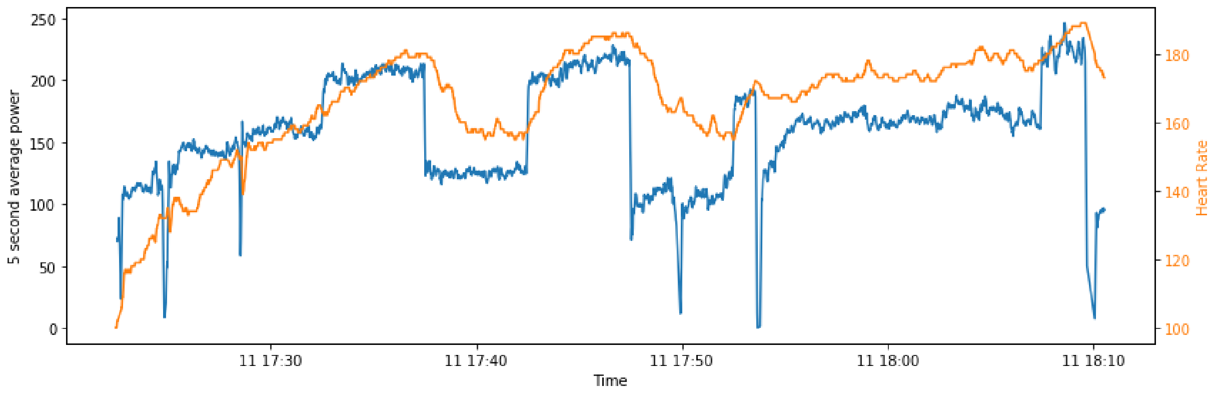
[11]
```
df_tcx['Distance'] = get_distance(df_tcx, from_speed=True)
df_tcx['DistanceMeters'] = df_tcx['Distance'] * 1000
df_tcx.head(4)
```

	Cadence	**DistanceMeters**	**HeartRateBpm**	**Speed**	**Watts**
Time					
2019-01-11 17:22:28	86.0	0.00	100.0	7.96	100.0
2019-01-11 17:22:29	86.0	7.88	100.0	7.88	97.0

	Cadence	DistanceMeters	HeartRateBpm	Speed	Watts
Time					
2019-01-11 17:22:30	85.0	15.74	100.0	7.86	97.0
2019-01-11 17:22:32	33.0	21.48	102.0	2.87	6.0

Let's have a look at these intervals:

```
[12]  df_interpolated = interpolate_to_second(df_tcx[['Watts',
      'HeartRateBpm']])
      dt = pd.to_timedelta(df_interpolated.index -
      df_interpolated.index[0]).values
      plt.figure(figsize=(12, 4))
      plt.plot(df_interpolated['Watts'].rolling(5).mean())
      plt.ylabel('5 second average power')
      plt.xlabel('Time');
      plt.sca(plt.gca().twinx())
      plt.plot(df_interpolated['HeartRateBpm'], c='C1')
      plt.ylabel('Heart Rate', color='C1')
      plt.gca().tick_params(axis='y', labelcolor='C1')
      plt.xlabel('Time')
      plt.tight_layout();
```



Calculating a Power curve

```
[13]  # TODO
```

Recovering Wattbike's relationship between Power and Speed

For a cyclist of weight W going up a gradient θ, with air density ρ, a coefficient of rolling resistance C_{rr}, drivetrain loss $\mathrm{Loss}_{\mathrm{dt}}$, cross-sectional area A and drag coefficient C_d,

$$P_{\text{legs}} = \left(1 - \frac{\mathrm{Loss}_{\mathrm{dt}}}{100}\right)^{-1} \left(gW(\sin\theta + C_{rr}\cos\theta) + \frac{1}{2}C_d A\rho V^2\right) V.$$

Or,

$$P_{\text{legs}} = \frac{gW(\sin\theta + C_{rr}\cos\theta)}{2\left(1 - \frac{\mathrm{Loss}_{\mathrm{dt}}}{100}\right)}V + \frac{C_d A\rho}{2\left(1 - \frac{\mathrm{Loss}_{\mathrm{dt}}}{100}\right)}V^3.$$

i.e.

$$P_{\text{legs}} = C_0 V + C_1 V^3.$$

Source.

Let's use `scikit-learn` to throw together a quick model which fits for power given speed, and see what wattbike thinks these coefficients are!

We'll fit both the $P = AV^3 + BV$ model above, and a general higher-order polynomial $P = \sum_{i=1}^{5} C_i V^i$ to see if there are some other terms Wattbike is using that isn't in the formula above. We fix $P_{\text{legs}}(V = 0) = 0$ as a boundary (i.e. don't fit an intercept).

We'll calculate a cross-validated score for each model and simply take the mean value as that model's score. We'll use negative median absolute error as a metric as it is more resilient to outliers (a higher score is better).

[14]
```
X = np.expand_dims(df_tcx['Speed'], 1)
X_simple = np.concatenate((X, X**3), axis=1)
X_predict = np.expand_dims(
    np.linspace(
        df_tcx['Speed'].min(),
        df_tcx['Speed'].max(),
        100
    ), 1
)
X_simple_predict = np.concatenate((X_predict, X_predict**3),
axis=1)

y = df_tcx['Watts']
```

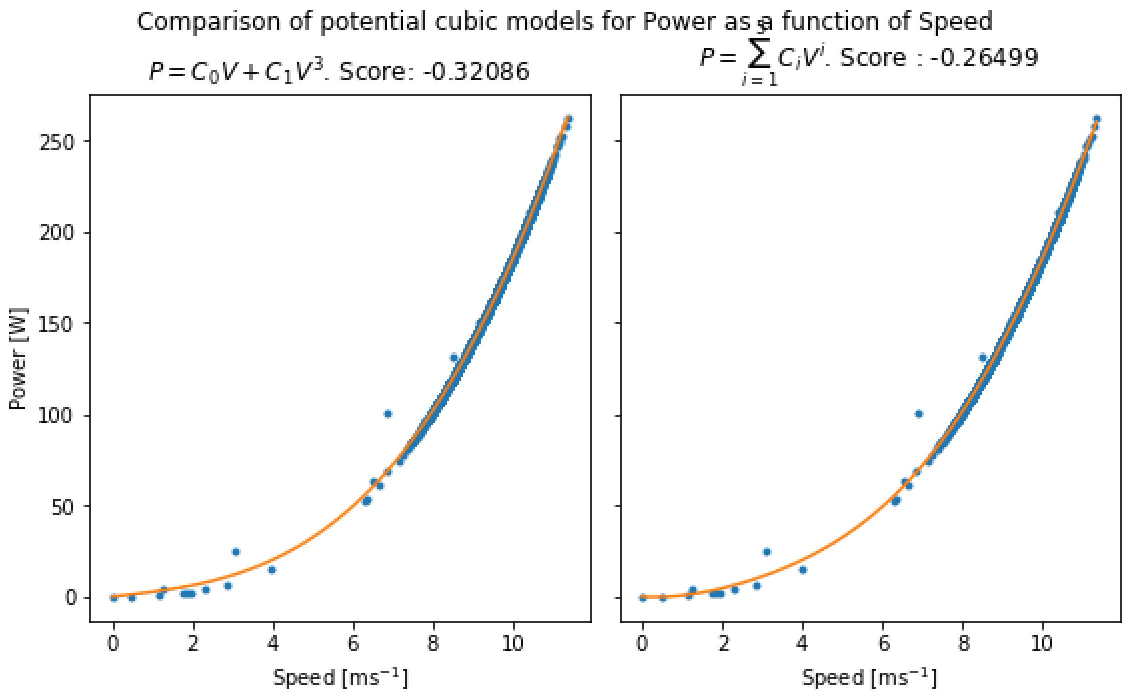
[15]
```
r_simple = LinearRegression(fit_intercept=False)
r_simple.fit(X_simple, y)
Y_simple = r_simple.predict(X_simple_predict)
```

```
score_simple = cross_val_score(r_simple, X_simple, y, cv=5,
scoring='neg_median_absolute_error').mean()
```

Ideally we would perform the complex fit using Bayesian Automatic Relevance Determination, which is better than simple OLS as it stabilises feature weights due to a sparsity-inducing prior (i.e. it likes to leave coefficients at zero), which is ideal for the question we are asking: whether certain coefficients should be zero.

However, `ARDRegression` takes a significantly longer time than OLS (and can cause memory problems), so I'm only mentioning it as an aside.

```
[16]    r_complex = Pipeline(steps=(
          ('features', PolynomialFeatures(degree=5, include_bias=False)),
          ('regressor', LinearRegression(fit_intercept=False)),
        ))
        r_complex.fit(X, y)
        Y_complex = r_complex.predict(X_predict)
        score_complex = cross_val_score(
            r_complex, X, y,
            cv=5, scoring='neg_median_absolute_error'
        ).mean()
```



The scores are almost identical between models, suggesting that our above model (P_{legs}) is a good match. This is despite the parameters of the models being very different.

```
Coefficient of P_legs model:
        P = 0.161 V^3 + 2.469 V
Coefficients of higher-order polynomial:
        P = -0.002 V^5 + 0.056 V^4 + -0.418 V^3 + 2.541 V^2 + -1.545 V^1
```

Why is this useful?

We now have a model for power as a function of speed that we can apply to our GPX data above (which doesn't contain power data as I can't afford a power meter!). Looking at the equation above for P_{legs} we see that we need to take into account the road gradient in the V coefficient, so our function will become

$$P_{\text{legs}} = \left[\frac{gW\sin\theta}{2\left(1 - \frac{\text{Loss}_{\text{dt}}}{100}\right)} + C_0\cos\theta \right] V + C_1 V^3,$$

$$P_{\text{legs}} = \left(\frac{C_0}{C_{rr}} \sin\theta + C_0\cos\theta \right) V + C_1 V^3.$$

Where C_0 and C_1 were fitted above.

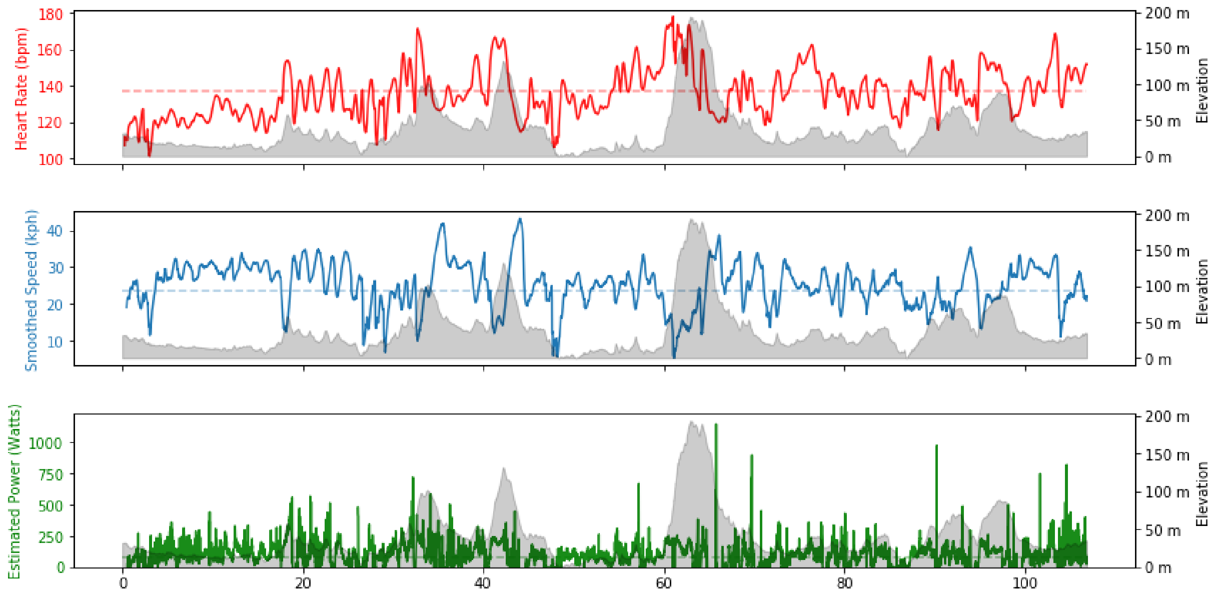
There would be some significant changes to other parameters detailed above as well, but as we have no means of measuring them it's best to assume they don't vary (we can experiment with using uncertainties to properly account for them later).

We will assume a value of $C_{rr} = 0.003$, which is consistent with those recorded here.

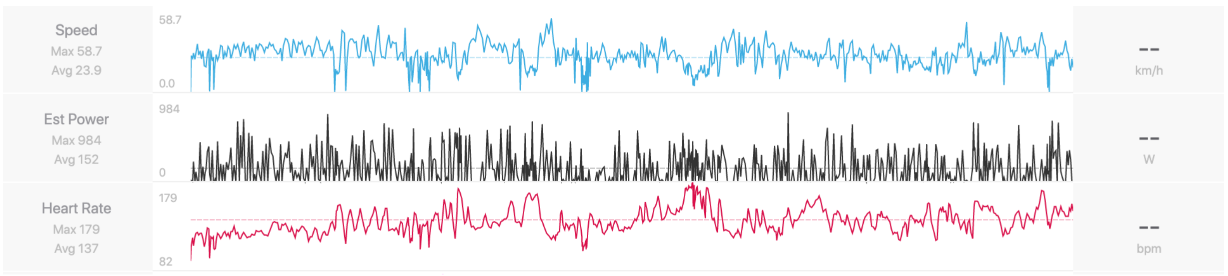
```
[19]  CRR = 0.003
      C_0, C_1 = r_simple.coef_
      gradient = get_gradient(df_gpx)
      theta = np.deg2rad(gradient)
      V = df_gpx['Speed'] * 10 / 36  # convert to metres per second
      (from kph)

      estimated_power = (C_0 / CRR * np.sin(theta) + C_0 *
      np.cos(theta)) * V + C_1 * V**3
      print(estimated_power.describe()[['mean', 'std', 'min', 'max']])
```

```
mean      80.203833
std       99.774665
min     -554.078839
max     1149.055423
dtype: float64
```

Now let's finish those charts from earlier!



For comparison, here's Strava's estimate:



It appears that this analysis has smoothed the data significantly more than they have, but the comparative values are still sensible!

It is very likely that Strava's model is significantly more comprehensive than the one presented here, given the volume of training data and number of analysists they have at their disposal, but the work here is a good first start! (it's also worth pointing out that the $C_d A$ assumed by Wattbike would be very optimistic compared to the true value of an amateur rider such as myself, so our analogy to the Wattbike model has some significant drawbacks).

Consistency checking

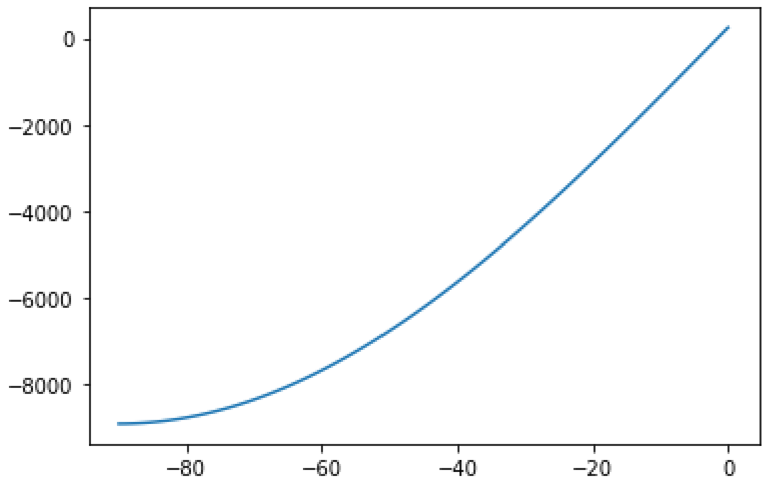
Observing the Power vs Elevation plot, we also see that on steep downhills the model is predicting a positive power output (i.e. it's assuming I'm having to push to get up to that speed despite the gradient, whereas I'm actually probably coasting). This might be a result of the boxcar smoothing used, or variations in the parameters of the P_{legs} model between the indoor and outdoor rides.

To investigate this, let's see what gradient I'd need to theoretically hold 40kph on a downhill without pedalling (something like 5% from experience)

```
[21]  V = 40 / 3.6
      theta = np.deg2rad(np.linspace(-90, 0, 100))
```

```
plt.plot(np.rad2deg(theta), (C_0 / CRR * np.sin(theta) + C_0 *
np.cos(theta)) * V + C_1 * V**3)
```

```
[<matplotlib.lines.Line2D at 0x1c22c420b8>]
```



So the model assumes I need a 60° downhill slope to go that fast! Something is definitely wrong there, we can investigate further in a different notebook.