Topics On Optimization and Machine Learning
Homework 3: Black Carbon Proxy

Adrià Lisa

Introduction

For this assignment we are asked to build a virtual sensor for the air concentration of Black Carbon (BC) using several machine learning techniques on a dataset of measurements of other pollutants and climate factors.

From now on, we will refer to every signal by their abreviated name in the dataset (BC, N_CPC, HUM, ...). The language used for coding our experiments was `python`, with heavy use of the libraries `seaborn` and `scipy`.

Data inspection and Correlations

The dataset that was provided contains a special column for the date of the measurements, another for some real values of BC, and 12 other numerical signals. In fig. 1, we display the visual representation of each of these 12 signals with our target BC.

The majority of the signals have positive correlation with BC. The only one with a negative correlation was the Ozone, O3. SO2 and TEMP have a negligible correlation, meaning that we can presume that they will not be of much use in the predictive models. The correlation with HUM is also quite weak, wich leads to the hypothesis that the climate factors (TEMP and HUM) are independent of BC.

In other words, the concentration of BC might only be related to the concentration of the other gases.

Among these 12, there are two that could be considered categorical data. SO2 only has 13 unique values, and CO has 23 unique values. This might just be due to limitations on the sensors for these pollutants, and since it is just easier for us to analyze them as numerical data, we will be doing so. We only treat them as categorical data for the representation in fig. 1.

Temporal trends

Before proceeding to explore the temporal trends of the data, we must normalize all the signals. A comparison on these terms of the raw data will not make sense, as the mean value of BC is 1.3, whereas the mean value for HUM is 70.8!. It goes without saying that the normalized data will also be the one that we use to train our machine learning models.

For every value of every signal X, with mean μ and standard deviation σ, we perform the following standard normalization operation:

$$X \leftarrow \frac{X - \mu}{\sigma} \tag{1}$$

Our data is not evenly distributed over time. Sometimes we have hourly measurements, but then we have intervals of several days without measurements. To cover this gap, we provide a visualization of the monthly-averages of every factor in figures 2 ,3 and 4.

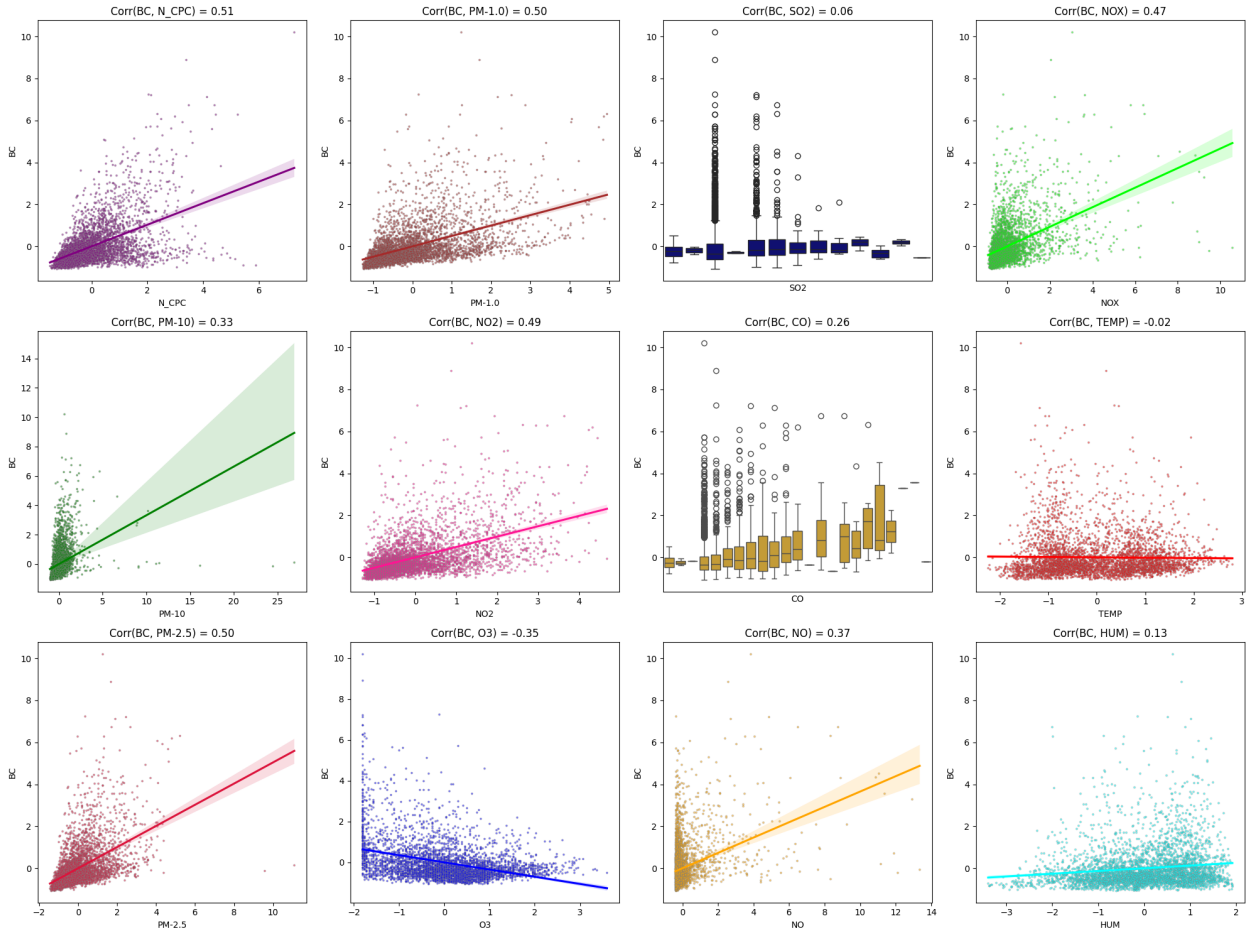

Figure 1: Relationships of each signal with BC. The regression lines were obtained using `seaborn.regplot`.

From fig. 2 we (roughly) see that the three **PM**s, along with the three **NO**s, are quite aligned with the trend of BC. Out of these 6 signals, it is clear that the one that diverges most from the trend is PM-10, the one that also has the lowest correlation.

The three signals from fig. 3 do not follow the trend from BC. Still, one may argue that O_3 follows an inverse trend as its correlation with BC is negative, but it is quite unclear.

For the last fig. 4, we represented the signals that have a different unit from BC, although in the end that fact is not very significant because all the measurements are normalized. Looking at the plot, it is not clear at all that any of the measurements follows the trend from BC, even though N_CPC has the highest correlation with BC among the other 12 signals.

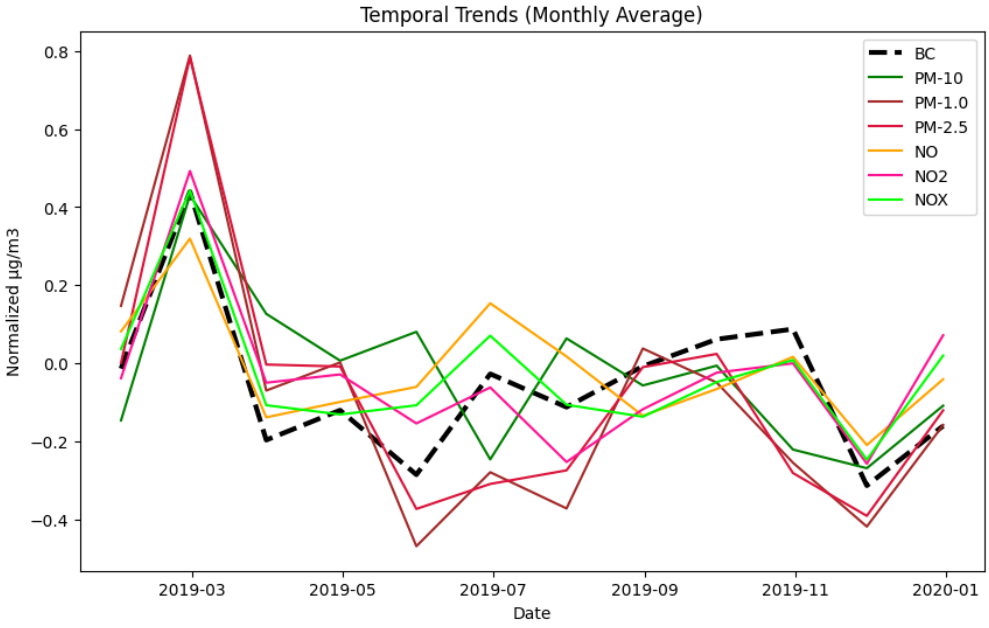

Figure 2: Monthly averages of the normalized gas measurements, with good correlation with BC.

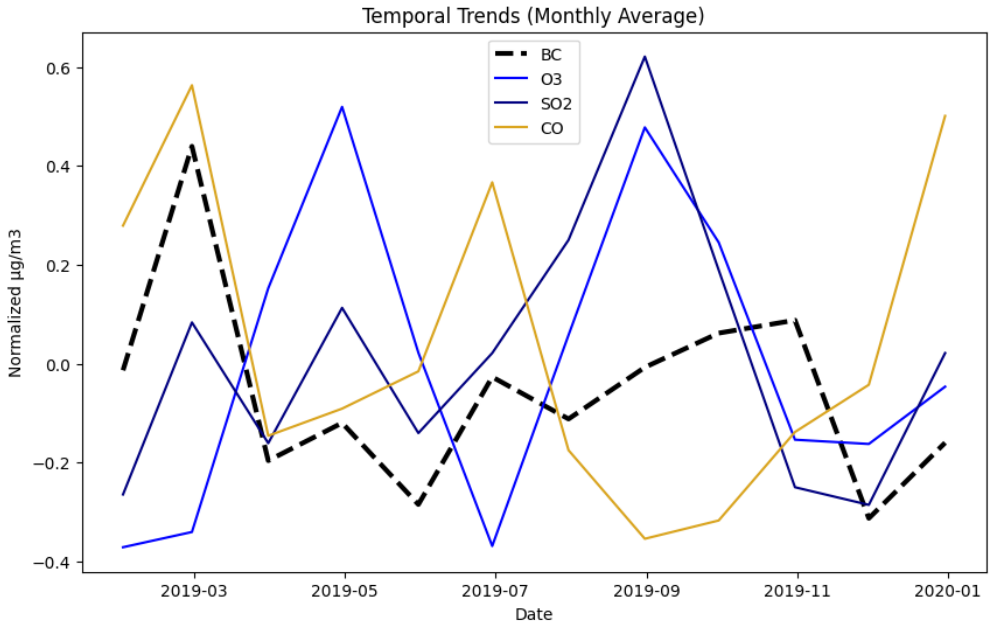

Figure 3: Monthly averages of the normalized gas measurements, with low or inverse correlation with BC.

Machine Learning Models

All the models will be predictors of the BC signal using the data from the other 12 signals. Therefore, they fill be functions form $\mathbb{R}^{12}$ to $\mathbb{R}$. We use 80% of the data to train our models, and the remaining 20% is used for testing. The testing consists on computing the coefficient of determination R^2 and the residual mean square error $RMSE$ between our predictions for the BC measure and the real values in the test subset.

Since none of the models will be using data from the past, intuition suggests that including information about temporality to our training process will not be useful. Our approach is to first build and tune our models using shuffled training data, which effectively drops the temporality information.

We used the `sklearn` library for the majority of the methods, and `torch` for the Neural Network.

Linear Regression (LinReg)

First, we did a Linear Regression model to serve as a baseline for the other models.

```
from sklearn.linear_model import LinearRegression

# Create a linear regression model
LinReg = LinearRegression()

# Train the model
LinReg.fit(X_train, y_train)

# Make predictions on the testing set
pred_linreg = LinReg.predict(X_test)
```

The syntax of `sklearn` is very simple, allowing us to build this model with ease. The resulting metrics for the linear

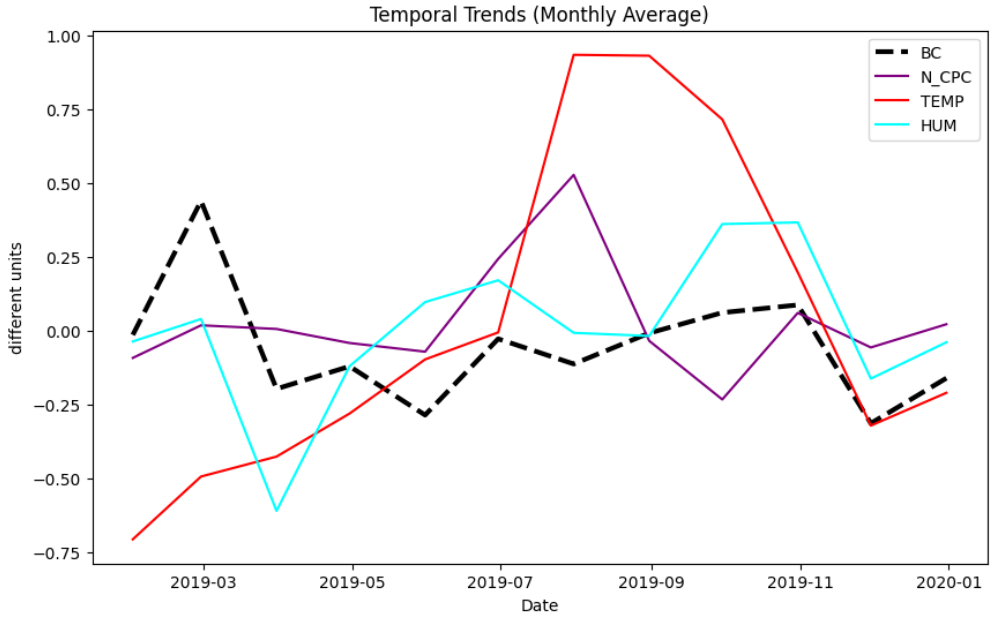

Figure 4: Monthly averages of the normalized signals BC ($\mu g/m^3$), N_CPC (unkwnown unit), TEMP (ºC) and HUM (%).

Kernel keyword	'linear'	'poly'	'rbf'	'sigmoid'
R^2	0.4817	0.5196	**0.6448**	-3414.7
$RMSE$	0.7305	0.7033	**0.6047**	59.3121

Table 1: Quick comparison between the `sklearn` kernels.

model were $R^2 = 0.5424$ and $RMSE = 0.6864$. A ML model with lower R^2 or higher $RMSE$ than this one can not be considered useful or correct.

Suport Vector Regression (SVR)

At first, we compared the different kernels that are implemented in the class `SVR` from `sklearn`, with the default values for the hyperparameters. We did not delve into how each kernel is actually defined, we just compared the results in table 1 and the obvious choice was the `rbf` or "radial basis function".

Still, the results for the 'sigmoid' kernel were very surprising, so we delved just a little on that. It is defined as $k(x, y) = tanh(\gamma \cdot x^T y + r)$. As it is not positive definite, the conditions of Mercer's theorem are not satisfied, which I believe might be the source of the issue.

In any case, a closer inspection at the empiric results revealed that this was a case of severe overfitting. The predictions in between data points oscillate heavily, but match the train values quite precisely at the data points. Therefore, the residual variance σ_r is extremely high, and since

$$R^2 = 1 - \frac{\sigma_r}{\sigma_y}$$

the value R^2 has a very large negative value.

The chosen kernel was the Gaussian radial basis function, which is defined as

$$k(x, y) = e^{-\gamma \|x-y\|_2^2}.$$

Therefore, we had to tune γ along with the regularization parameter C and the "tube width" ε. For that, we used the very convenient `GridSearch` routine. We only tested values for each hyperparameter at logarithmic scale, as can be seen in the following snippet:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR

# Define the parameter grid
param_grid = {
    'C': [0.1, 1, 10, 100],
    'epsilon': [0.01, 0.1, 1],
    'gamma': ['scale', 'auto', 0.001, 0.1, 1, 10]
}

svr = SVR(kernel='rbf')

# Instantiate the grid search model. Uses all processors with "n_jobs=-1"
grid_search = GridSearchCV(estimator=svr, param_grid=param_grid, n_jobs=-1)

grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
```

The values that attained lower training loss were $C = 10$, $\varepsilon = 0.1$, $\gamma =$`'auto'`. The keyword `'auto'` sets the value to one over the number of features, which for our case would be $\gamma = 0.0833$.

Using this hyperparameters, the SVR model test metrics were $R^2 = 0.7097$ and $RMSE = 0.5467$, considerably better than the values for the RBF kernel in table 1.

Random Forest (RF)

The `RandomForestRegressor` bootstrapping ensemble method from `sklearn` with default parameters had better results than the tuned SVR, namely, an $R^2 = 0.796$ and $RMSE = 0.5467$.

As there were no performance issues with this model, we saw no point in limiting the depth of the trees. It was also clear that bootstrapping was beneficial in general.

We considered tree hyperparameters governing the construction of every tree, namely:

- `max features`: the number of features to consider when looking for the best split. We could simply choose all of them, without the performance dropping too much. That was the default value.
- `min samples split`: the minimum number of samples required to split an internal node. The best was the default value, 2.
- `min samples leaf`: This is the minimum number of samples required to be at a leaf node. The best was the default value, 1.

Then, we conducted an exploration on the best number of trees in the forest, M, whose default value is $M = 100$. Naturally, when increasing M there is a trade off between expected accuracy and speed, but there must be a point where the accuracy settles. This hypothesis was correct, as figure 5 suggests.

Since this model is non deterministic, each time it was defined and trained the resulting metrics would be different. We repeated the training-testing process 100 times, to compute some statistics on the test metrics. We also took into

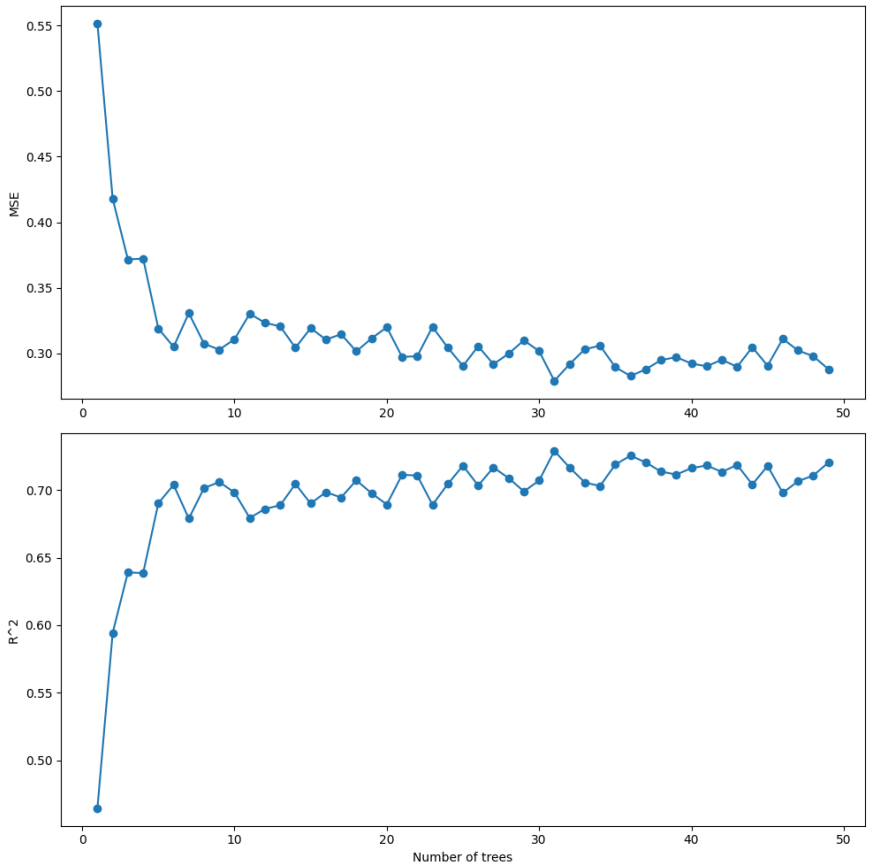

Figure 5: Test metrics samples of the Random Forest Regressor w.r.t. the number of trees.

account the differences in speed by measuring the time of this process, and they proportional to M, unsurprisingly. All the measurements are referenced in table 2

nº of trees M	1	10	50	100	200
Mean R^2	0.4356	0.6949	0.7171	0.7201	0.7211
Std R^2	0.0614	0.0146	0.0054	0.0044	0.0039
Mean $RMSE$	0.7612	0.5603	0.5396	0.5368	0.5359
Std $RMSE$	0.0407	0.0134	0.0052	0.0042	0.0038
Speedup	85.59	9.51	1.98	1	0.59

Table 2: Statistics over 100 trials on the results of RF, with respect to different values for the number of trees M. The speedup metric is not a precise measurement as there is some time-overhead when computing the statistics.

As performance is not really an issue for this academic project, we settled on choosing $M = 200$. However, in a scenario where real time measurements are expected, M could be dropped to 10 for a $\times 20$ speedup without much accuracy losses.

Feed-Forward Neural Network (FNN)

There is a vast ocean of architectures for Neural Networks. We restricted our sights on the multi-layer perceptron model, with only one hidden layer of arbitrary width. The input dimension is the number of features, 12, and the output dimen-

sion is 1 corresponding to our 'BC' prediction. The framework we used was `PyTorch`.

We started by using the simplest activation function, the `ReLU`. To decide on an appropiate width, we started by iterating on all possible values in the range $w \in 1, \ldots, 100$, with a very simple training porcess:

```
import torch
from torch import nn

# Iterate over different network widths
for width in range(1, 101):
    Fnn = nn.Sequential(
        nn.Linear(X_train.shape[1], width),
        nn.ReLU(),
        nn.Linear(width, 1)
    )

    # Define the loss function and the optimizer
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(Fnn.parameters(), lr=0.01)

    # Train the model
    for epoch in range(50):
        Fnn.train()
        optimizer.zero_grad()

        y_pred = Fnn(X_train_torch)
        loss = criterion(y_pred, y_train_torch.view(-1, 1))

        loss.backward()
        optimizer.step()
```

The results for the test metrics for every width w can be seen in figure 6. As in the analysis for M in the RF model, there is some stochasticity in the results, but they end up settling.

Just as a curiosity, we performed the same analysis changing the number of training epochs from 50 to 100. See figure 7. Naturally, the metrics were better, and they also had less stochasticity.

Since at this academic project we are not constrained by performance, we could have set $w = 1000$ and the training/testing process would still be very fast. However, that would be senseless, as the number of parameters of the network would surpass the number of data points. In the end, we decided to settle on $w = 100$, because the resulting network had 1401 parameters, approximately a half of the amount of data points we were working with in the train sample.

Afterwards, we shifted our focus on the activation function. Having the structure of the network fixed, we run a routine that compare the test results using some of the most popular activation functions in the `Pytorch library`. We used 100 training epochs, and since the stochasticity was to be low, we only took 10 samples of the testing metrics. The resulting statistics are compiled in table 3.

The top 4 activation functions were `ReLU` and `LeakyReLU`, which are well known, and `PReLU`, which were unkown to the author. After an inquiry, we discovered that `PReLU` is actually equivalent to `LeakyReLU`, but with the leak parameter being learned.

We ended up choosing `PReLU`, as it just had the best results, and it only adds one parameter to the network.

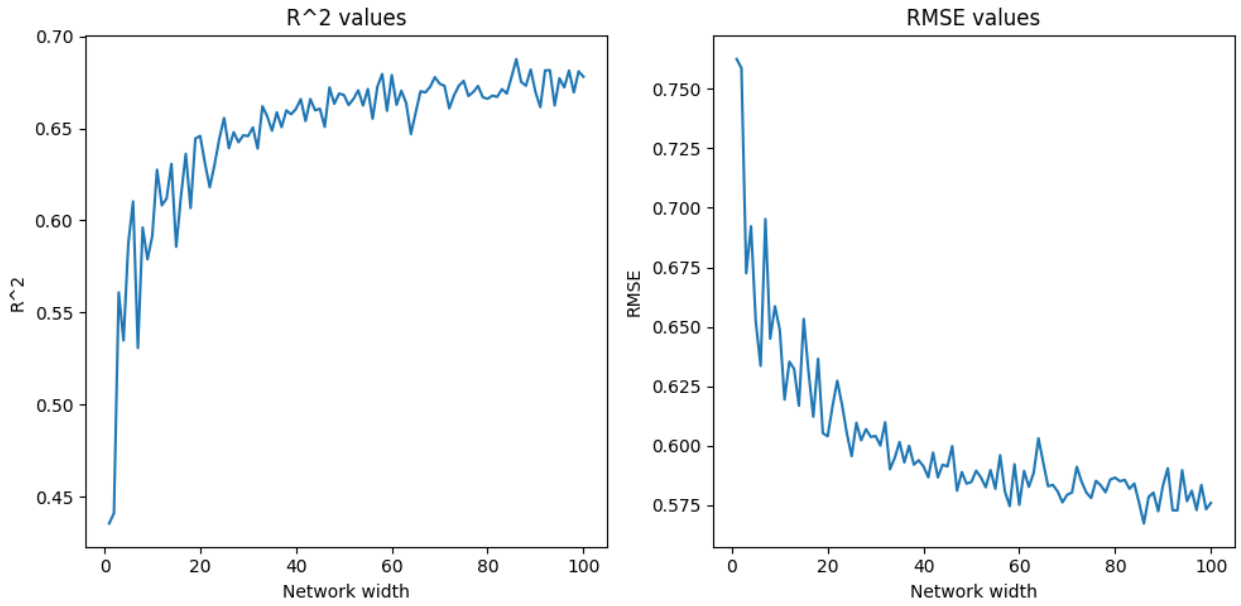

Figure 6: Test results compared w.r.t. the network with, with a training porcenss of 50 epochs.

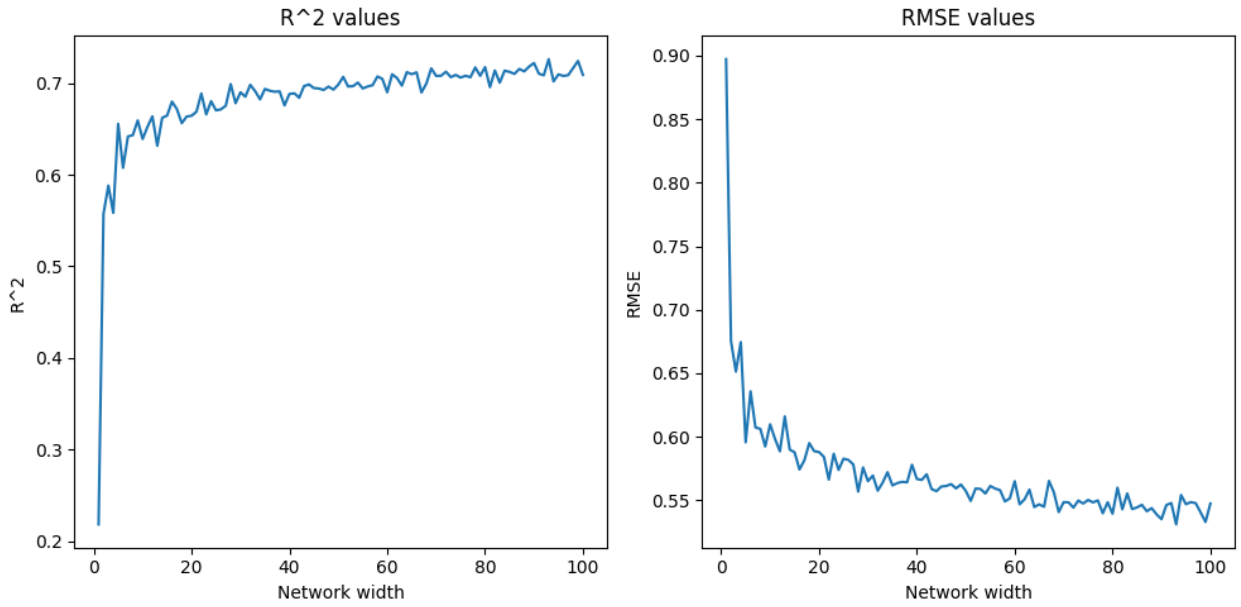

Figure 7: Test results compared w.r.t. the network with, with a training porcenss of 100 epochs.

Activation Function	Mean R^2	Std R^2	Mean $RMSE$	Std $RMSE$
ReLU	**0.7139**	0.0064	**0.5427**	0.0060
Tanh	0.6933	0.0083	0.5619	0.0076
Sigmoid	0.5802	0.0093	0.6575	0.0073
ELU	0.6798	0.0030	0.5743	0.0027
PReLU	**0.7288**	0.0082	**0.5285**	0.0080
LeakyReLU	**0.7145**	0.0085	**0.5422**	0.0080
Hardtanh	0.6975	0.0100	0.5581	0.0092
SELU	0.6789	0.0103	0.5750	0.0092
CELU	0.6824	0.0066	0.5719	0.0059
GELU	0.7011	0.0043	0.5548	0.0040
SiLU	0.6924	0.0034	0.5629	0.0031
Mish	0.6918	0.0067	0.5633	0.0061

Table 3: Statistics over 10 trials on the results of different activation functions.

Conclusion

Since the stochasticity in some of our models only came from the training processes, we can now give the results of the trained, final models in a single table 4. We also measure the speed of making a prediction, relative to the speed of the linear model.

Final Trained Model	**R²**	**RMSE**	**Speedup**
Linear Regression	0.5425	0.6865	1
SVR	0.7097	0.5468	95
Random Forest	0.7253	0.5319	20
Neural Network	0.7148	0.5419	0.5

Table 4: Compared results between the models described in the previous section, with data shuffling.

All three models achieved qualitatively similar results, with the Support vector Regression being relatively inferior. It has to be noted that the performance boost of the FNN is due to it being implemented in `Pytorch`, and it has to be taken with a grain of salt as the network has 1402 parameters. The other 3 methods are all implemented in `sklearn`, so the speedup comparison is fair. The final Random Forest has $M = 200$ trees, so we can deduce that with $M = 10$ trees it will have approximately the same running time as the Linear Regression.

To shuffle, or not to shuffle?

We re-trained our final models with a training set that is not shuffled. The results in table 5 are considerably worse that our previous results. This confirms the hypothesis that for this case it is better to shuffle the data.

Model	**R²**	**RMSE**	**Speedup**
Linear Regression	0.5277	0.6850	1
SVR	0.6872	0.5575	94.5
Random Forest	0.6341	0.6029	18.9
Neural Network	0.6868	0.5578	0.5

Table 5: Results having the train data not be shuffled

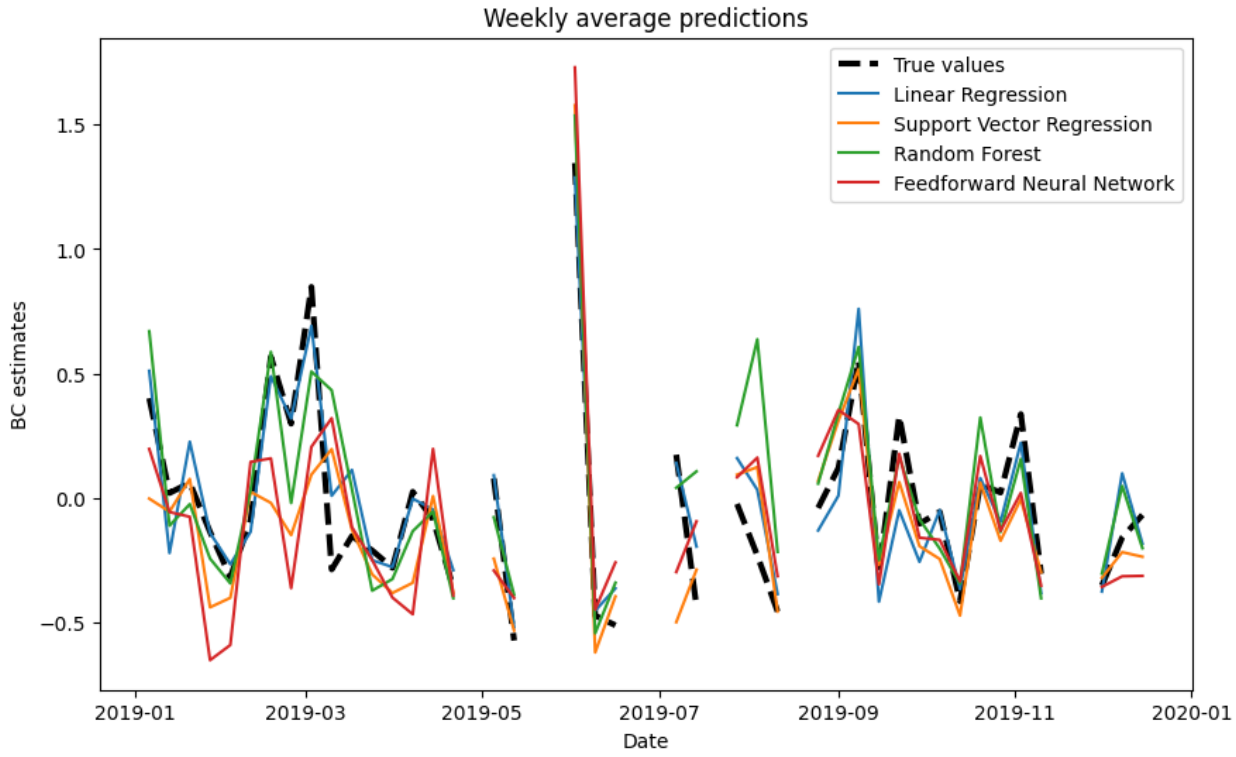

Figure 8: BC averages over a week. The gaps are due to whole weeks being absent in the dataset.

Visulization vs the dates

To conclude, we provide some plots on our model predictions versus the date in figures 8 and 9. For visualization purposes, we can only show the averages over a time span, as the information of the dataset is not homogeneous over time.

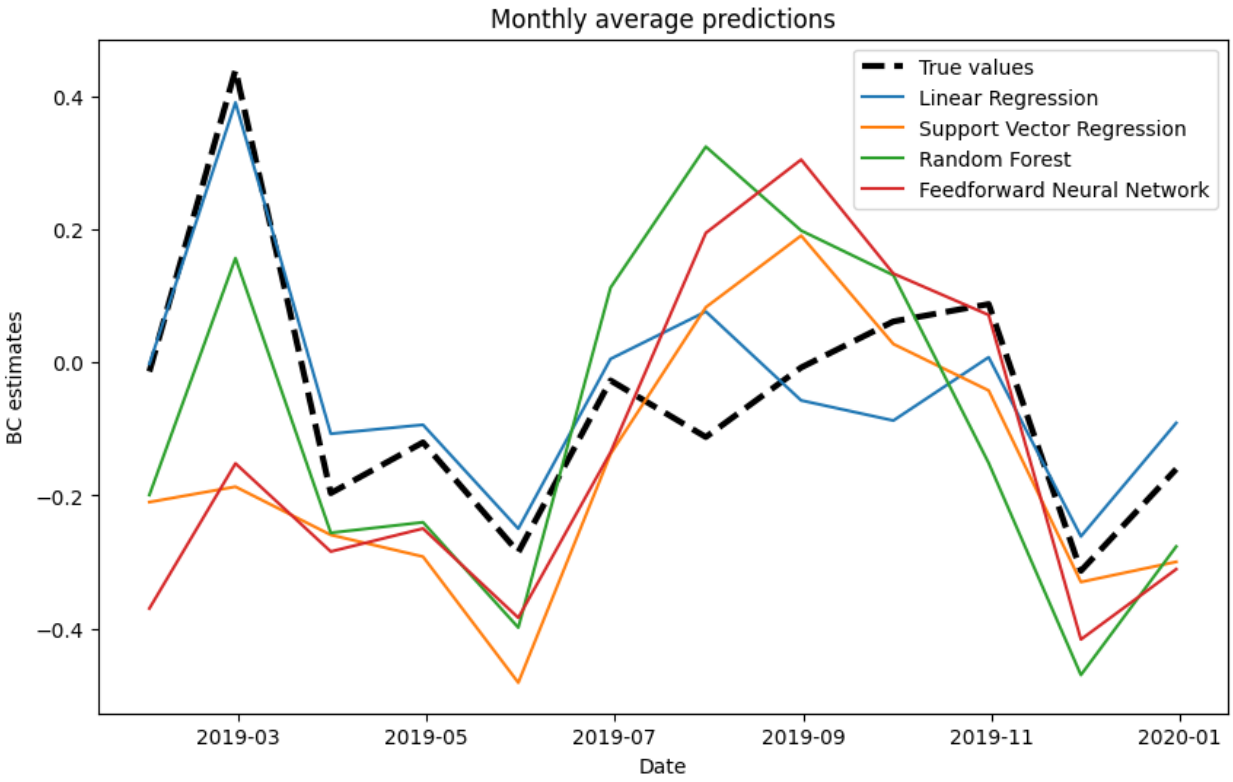

Figure 9: BC averages over every month in the dataset.