ML – Fusion

Contents

[Aktualno 2](#_Toc145238499)

[To-Do: 3](#_Toc145238500)

[Kaj ne gre: 4](#_Toc145238501)

[Struktura repozitorija 4](#_Toc145238502)

[OLD 5](#_Toc145238503)

[Povzetek 5](#_Toc145238504)

[TO-DO: 5](#_Toc145238505)

[Start 5](#_Toc145238506)

[Training models for each x\_m. 5000 models. 9](#_Toc145238507)

[Machine learning process – for paper 9](#_Toc145238508)

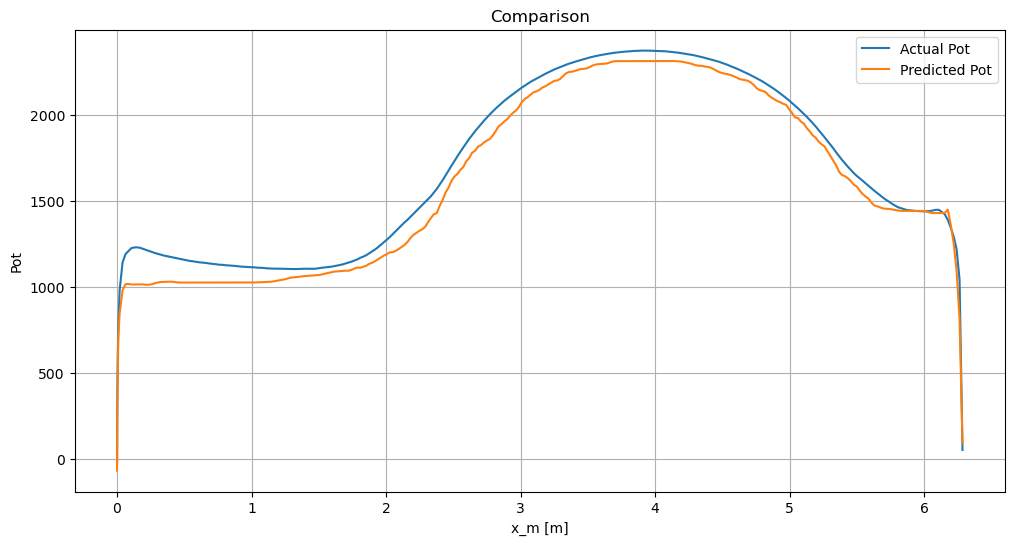
[XGBoost 9](#_Toc145238509)

[Drugi modeli 16](#_Toc145238510)

# Aktualno

1. Prej so bili najboljši rezultati tam kjer je en model za vsak x\_m. To sem sedaj preizkusil in izboljšal z drugim konceptom. Ta koncept je tunanje hyperparametrov s tem, da se testira z Leave one out metodo - LOO. Pri trenutnih rezultatih smo imeli kar kratko optimizacijo (144 kombinacij). Bi lahko bilo mogoče še boljše, ampak najverjetneje ne. Za tako število kombinacij kot jih imamo (21) je to že kar vredu rezultat.
2. Tu smo sedaj ugotovili, kaj je najboljše. Ponovimo še za druge parametre z isto metodo.

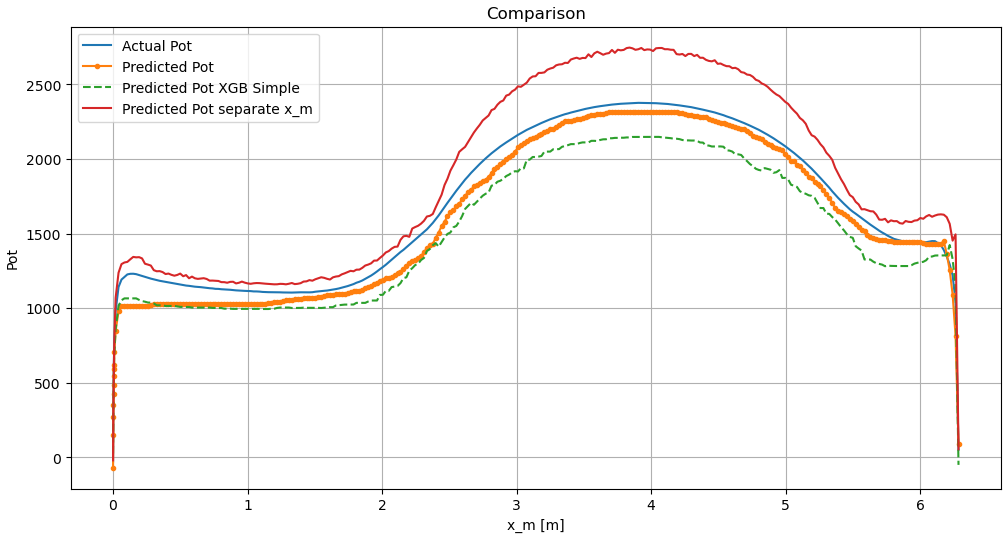
Napoved in actual podatki za vse spremenjene parametre. Mean absolute percentage error je tu 6,5 %. Predicted pot je tu narejena z xgb-LOO.



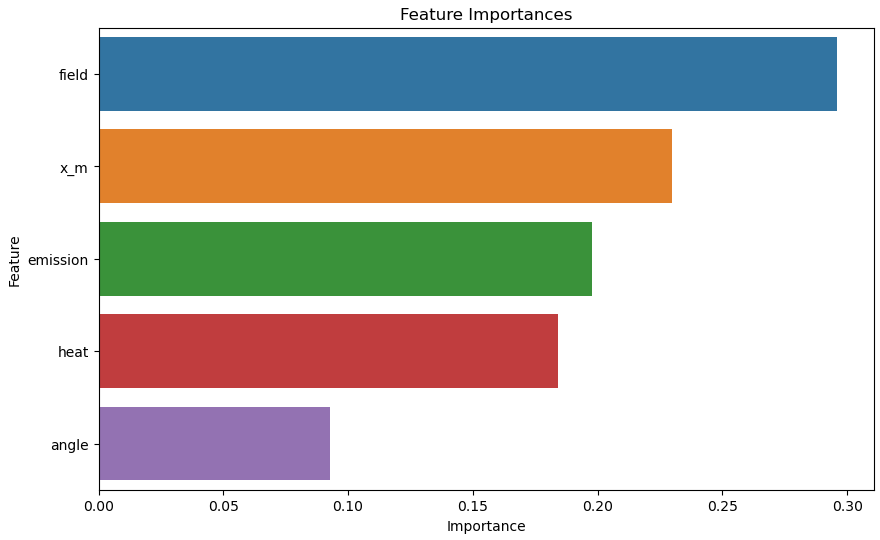
Spodaj vidimo različne pristope. Modro je actual. Oranžno je xgb-LOO. Z zeleno je označen naiven xgb model, ki se nauči na vseh podatkih enakomerno. Vidimo, da smo z LOO pristopom izboljšali natančnost. Z rdečo pa je prikazan pozkus, kjer je za vsak x narejen svoj model. Vidimo, da je kar velika papaka. Je pa kar skoraj presenetljivo, da še vedno dobimo tako zvezen graf.

Ugotovimo lahko, da smo prišli do metode, ki nam bo prinašala najboljše rezultate. To ponovimo še za druge fuzijske ciljne spremenljivke. Uporabljena metoda pa tudi ni čisto klasična in bo zanimiva za v članek.

To ni klasičen primer, kjer razdeliš podatke na train in test set, naučiš (še cross validacija) in potem preveriš. Zato, ker imaš v tem primeru pomešane podatke iz testa že v train. Tako dobiš »predobre« rezultate (data leakage). Ko pride do novih podatkov so pa slabi rezultati. Tako da tukaj tudi ne pride v poštev navaden grid search. Če bi imeli večje število kombinacij pa bi lahko tudi po klasični poti verjetno prišli do boljših rezultatov.

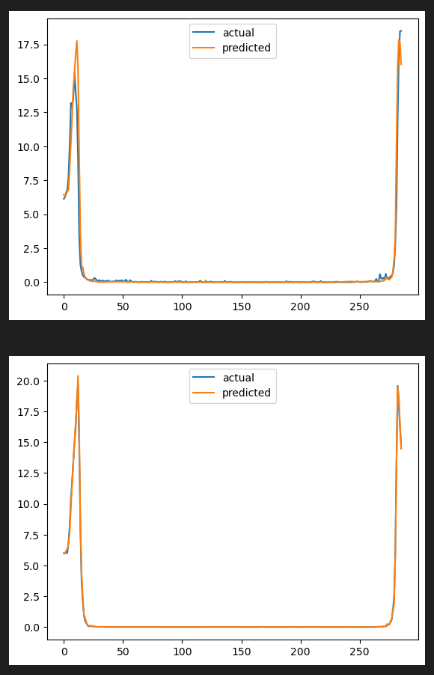


Feature importance za najboljši model xgb-LOO



## To-Do:

1. Naredi še za druge parametre
   1. Nič pametnega: E, nn, Ve, E
   2. Nardit: ne/ni, Te, Ti, Tn, Vi, Vn?,

Za Tn tudi potencial, da bo vredu na ta način. 

## Kaj ne gre:

* Z GAN ne gre
* Potestiral sem še s timeseries modeli – SARIMAX. Ni vredu.
* Osnovne neural networks slabše
* Vredu bi lahko bile recurring neural networks – sem delno preveril ampak ne do konca (nekje med skriptami v mapi potential\_analysis. Tudi slabše.

## Struktura repozitorija

**data**

1. Output\_fusion.xlsx ->
   1. df\_data.csv (transformirano z *extracting\_data.py*)
      1. shranjeno v data\_cleaned\_sparse.xlsx za hitrejše učenje
2. new\_output\_fusion -> nove simulacije
3. Tested\_further\_Te – najboljši parametri – smo najdli v skripti *separate\_models\_xm\_xgb\_finding.py*

**Skripte**

1. xgb\_LOO\_params.py iskanje najboljših parametrov za xgboost – leave one out
2. xgb\_model – trenutno stanje pri analizi. Primerjava različnih modelov

# OLD

## Povzetek

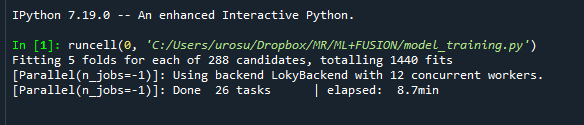
Trenutno so najboljši rezultati z XGBoost modelom. Za vsak x svoj model. Povprečna relativna napaka je 11 %. Težko bo veliko izboljšati glede na to, da imamo samo 21 primerov, iz katerih se model lahko uči. Delno lahko še izboljšamo še z domenskim znanjem, vemo, da parameter Pot ne more biti negativen. Poleg tega lahko izločimo še “outlierje” in zgladimo izgodni graf. Vidno so pod Slike.

### TO-DO:

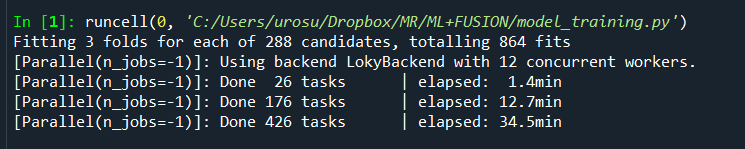
* Upoštevati še to domensko znanje.
* Odstraniti outlierje in zgladiti izhodne grafe
* Določiti najboljše hyperparameter še za druge tarčne parametre.
* Ponovno stestirati z združenimi podatki, kjer se nauči samo en model. Podoben poizkus kot s tistimi 5000imi modeli.

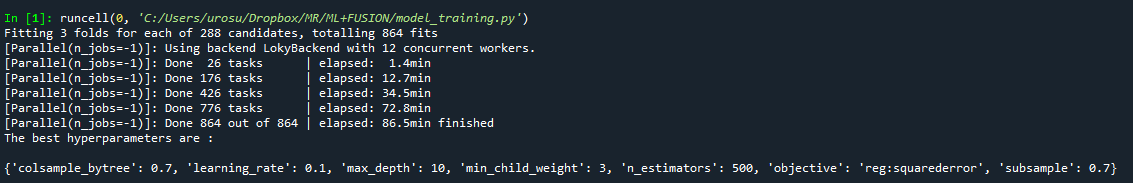
## Start

1. Najprej sem pripravil podatke za učenje modela. To je v file extracting\_data.py. Tam pretvorimo neurejene podatke v excelu v podatke primerne za učenje modela.
2. Na začetku sem poizkusil na tak način, da sem za featurje vzel, angle, heat, field in emission. **Za target pa posamezne parametre – na primer »Pot«. Tako sem moral za vsak x naučit svoj model. Torej nekje 120k modelov. S tem so problemi, ker je manj podatkov za učenje modela in tudi problem z uporabo naučenih modelov.**
3. **Odličil sem se, da bo najboljše, da med featurje dodam tudi razdaljo po tokamaku, torej x [m].** Tako dobimo za vsak x 21 različnih kombinacij parametrov. Skupno je tega 2 500 000.
4. Nato sem začel z učenjem modelov. Za prvi target feature, kjer bom preizkušal modele sem izbral parameter »Pot«. Začel sem z ZELO popularnim drevesnim modelom xgboost. Z osnovnimi nastavitvami je model vrnil napovedi, ki so se natančno skladale z enim ali drugim setom podatkov za učenje. V bistvu ni vrnil novih vrednosti ampak je vrnil krivuljo, ki je najbolj podobna enemu setu podatkov iz učenja.
5. Tega nismo iskali… Zato sem najprej probal najti najboljše hyperparametre. To sem delal z metodo GridSearchCV. Najprej sem probal z velikim številom podatkov in veliko variacijo hyperparametrov. To bi trajalo kakšnih 10 ur.

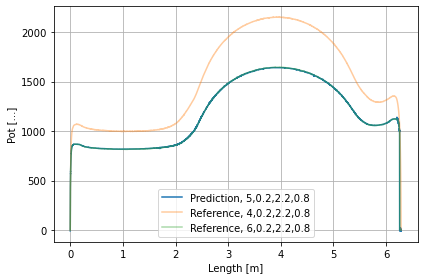


5 folds za 288 kombinacij naših parametrov.

Zato sem zmanjšal podatke na katerih poizkuša. Vzame 20% naključnih vrstic in poizkuša na tem.  
To traja nekje 1 uro.  


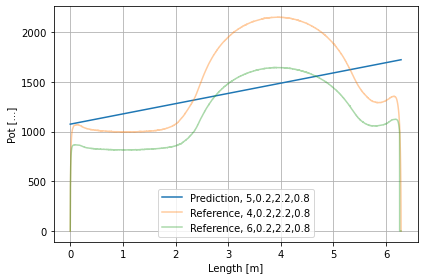
  
  
Parametri so: {'colsample\_bytree': 0.7, 'learning\_rate': 0.1, 'max\_depth': 10, 'min\_child\_weight': 3, 'n\_estimators': 500, 'objective': 'reg:squarederror', 'subsample': 0.7}  
RMSE: 6.524884

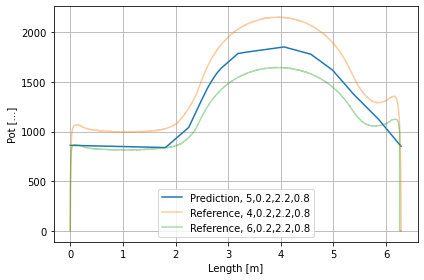
Training score: 0.9998815920732039



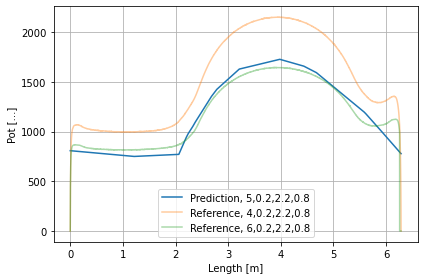
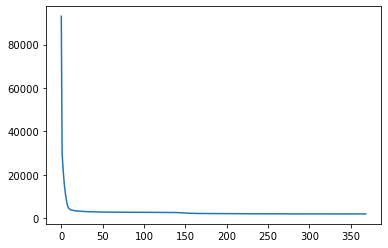
1. To še vedno ni vredu. Zato sem poizkusil z linearno regresijo.

To žal ne bo delalo, ker imamo x- Length za parameter. Bo treba drugače zapakirat? Tako kot sem začel v prvi točki? Za vsak x svoj model?

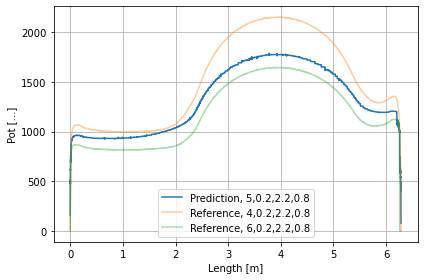


1. Test z nevronsko mrežo. 20% data, 500 iteracij  
     
   C:\ProgramData\Anaconda3\lib\site-packages\sklearn\neural\_network\\_multilayer\_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (500) reached and the optimization hasn't converged yet.
2. Test z nevronsko mrežo. 100% data, 1000 iteracij

Iteration 369, loss = 2074.93204292

Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.  
  


**Iskanje hyperparametrov v xgboost**



colsample\_bytree = 0.3, learning\_rate = 0.25, max\_depth = 5, min\_child\_weight = 3, n\_estimators = 300, subsample = 0.7

Potrebno drugače strukturirat podatke.

<https://colab.research.google.com/drive/1oZaL-hwSIQ0UU78sFjiK8LeuTUn1xBYa#scrollTo=xupigeWN-PMw>

Tu sem učil ensamble za max pot lokacijo.

Test xgboost gridsearch za max pot en x\_m

xg\_reg = xgb.XGBRegressor(

objective = 'reg:squarederror',

colsample\_bytree = 0.5,

learning\_rate = 0.9,

max\_depth = 1,

min\_child\_weight = 1,

n\_estimators = 200,

subsample = 0.1)

## Training models for each x\_m. 5000 models.

Mislim, da prejšnji princip kjer so vsi podatki združeni ni bil vredu. Zato sem ponovno poizkusil z učenjem posebnega modela za vsak x.

### Machine learning process – for paper

Creating a machine learning model typically involves the following steps:

* Define the problem and determine the appropriate type of model.
* Prepare the data by cleaning, formatting, and splitting it into training, validation, and test sets.
* Choose and implement a machine learning algorithm that is suitable for the problem and data.
* Train the model on the training data, using the implemented algorithm.
* Evaluate the model's performance on the validation data, and use this feedback to fine-tune the model, such as by adjusting the hyperparameters of the algorithm.
* Test the final model on the test data to assess its performance on unseen data.
* Deploy the model in a production environment, and monitor its performance to detect and address any issues.

It is important to note that some of these steps may be iterated several times, such as fine-tuning and testing, before arriving at a final model that is suitable for the problem. Also, in some cases, feature engineering is done before training the model to improve the model's performance.  
In our case no feature engineering could be done.

In this study, we first prepared and exported the data using the script 'extracting\_data.py'. We then trained various models on the train data, and found that the best model was the xgboost algorithm. However, we encountered issues with overfitting due to the structure of the train data. To address this, we set up the training process to create a new machine learning model for every x on the length of the tokamak, resulting in approximately **5942** models. Each model was trained on 20 datapoints and tested on one.

### XGBoost

To improve the performance of the xgboost model, we tuned the hyperparameters through 5000 tests. Initially, 8800 tests were planned, but the working day was over. The best hyperparameters were determined, specifically for the target Pot. The relative error of the fitting was found to be 5%.

XGBoost (eXtreme Gradient Boosting) is an open-source library for gradient boosting on decision trees. The library provides a powerful and efficient implementation of the gradient boosting algorithm. It has several hyperparameters that can be adjusted to optimize the performance of the model:

* **learning\_rate:** This is the step size shrinkage used to prevent overfitting. Smaller values generally result in models that are more robust, but may be slower to train.
* **n\_estimators**: This is the number of trees in the model. Increasing this value will improve the model's performance but will also increase the training time.
* **max\_depth**: This is the maximum depth of each tree in the model. Increasing this value will make the model more complex and may lead to overfitting.
* subsample: This is the fraction of the training data used for each tree. Smaller values will make the model more robust but may decrease its performance.
* **colsample\_bytree**: This is the fraction of the features used for each tree. Smaller values will make the model more robust but may decrease its performance.
* **min\_child\_weight**: used to control the minimum sum of instance weight(hessian) needed in a child. This is used to prevent overfitting by avoiding the creation of child nodes with a small number of samples. When the sum of the instance weight of the child is less than min\_child\_weight, the tree splitting stops. In other words, this parameter defines the minimum sum of the hessian matrix required for a split to happen in a tree. Hessian matrix is a second-order derivative matrix and is calculated using the gradient of the loss function with respect to the model parameters. The larger the value of min\_child\_weight, the more conservative the algorithm will be. This means that it will be less likely to split a node with a small number of samples. Setting the min\_child\_weight to a higher value will make the model more robust but may decrease its performance. On the other hand, setting it to a lower value will make the model more complex and may lead to overfitting. The optimal value of this parameter depends on the problem, the dataset, and the other hyperparameters of the model
* **Subsample**: controls the subsampling of the training data before growing each tree. It is a fraction of the training data that is used for each tree. The remaining data is not used in the model, and this is called "stochastic" or "random" subsampling. By subsampling the data, XGBoost is able to train each tree on a different subset of the data, which can help to reduce overfitting. When subsampling is set to 1, all of the training data is used for each tree, and when it is set to a value less than 1, a random subset of the training data is used. A smaller value of subsample will make the model more robust but may decrease its performance. On the other hand, a larger value will make the model more complex and may lead to overfitting. The optimal value of this parameter depends on the problem, the dataset, and the other hyperparameters of the model.

It is also important to note that subsampling can be combined with feature subsampling to achieve even better results. In feature subsampling, a random subset of the features is used for each tree. This can further reduce overfitting and improve the model's robustness.

* reg\_alpha: This is the L1 regularization term on weights. It can help to reduce overfitting.
* reg\_lambda: This is the L2 regularization term on weights. It can help to reduce overfitting.
* objective: This is the objective function to be minimized. XGBoost supports various objectives like binary:logistic, multi:softmax, etc.
* booster: This is the type of booster. XGBoost supports various boosters like gbtree, gblinear, dart, etc.
* n\_jobs: This is the number of parallel threads used to run xgboost. It can be used to speed up the training time.

These are some of the most common and important hyperparameters of the XGBoost model. However, it has many more hyperparameters that could be used for fine-tuning the model performance based on the problem and dataset.

#### Training XGBoost

**This was determined on data that was sampled 1:200. The training and output on all models takes about seven minutes.**

BEST HYPERPARAMETERS

xg\_reg = xgb.XGBRegressor(

objective = 'reg:squarederror',

learning\_rate=0.1,

max\_depth = 3,

min\_child\_weight = 3,

subsample = 0.7,

colsample\_bytree = 0.6,

n\_estimators = 900)

BETTER

xg\_reg = xgb.XGBRegressor(

objective = 'reg:squarederror',

learning\_rate=0.1,

max\_depth = 1,

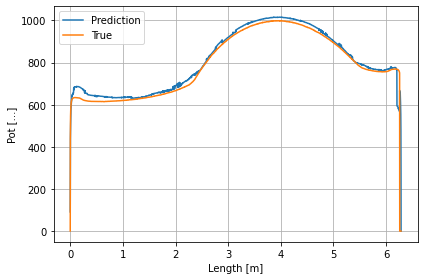
min\_child\_weight = 1,

subsample = 0.1,

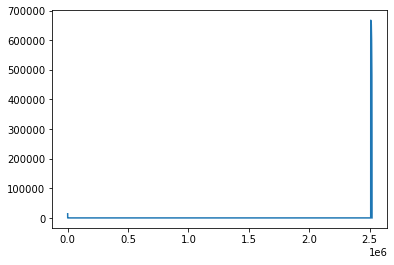
colsample\_bytree = 0.1,

n\_estimators = 500)

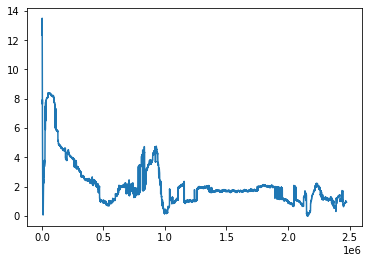
“Pot” PLOT ALL DATA s temi better hyperparametri. To je na 8 - data.iloc[8::21, :]



Relativna napaka po točkah.



Če izpustimo prvih 100 (0.0052662 m) in zadnjih 100 (6.1783 m). V procentih.



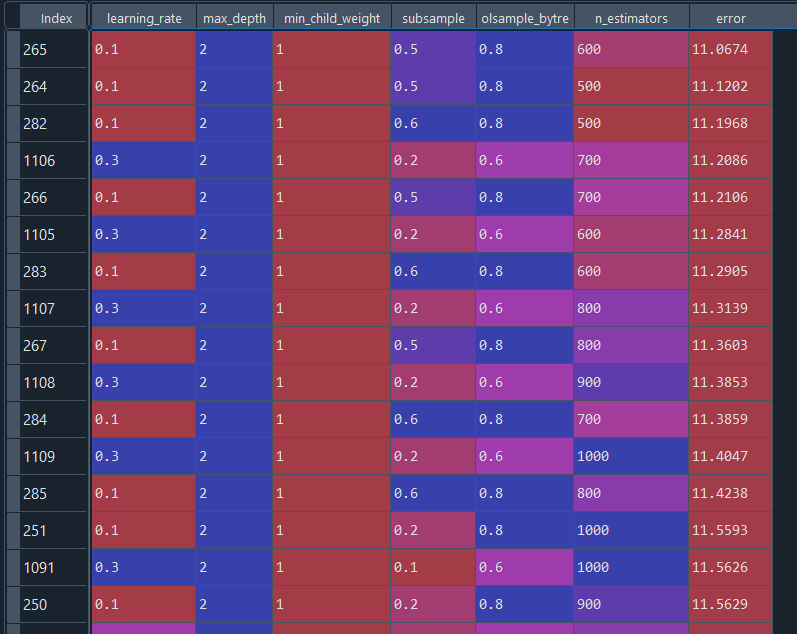
**Povprečje 2.23 %**

**Problem, da sem optimiral za samo eno zadevo. Moram pociklat po vseh in vzet povprečje in najmanjšo vrednost!**

#### Finding smallest average relative error across all instances

1296 kombinacij – za vse, povprečna relavitna napaka

Spodaj so prikazani nekatere najboljše kombinacije parametrov



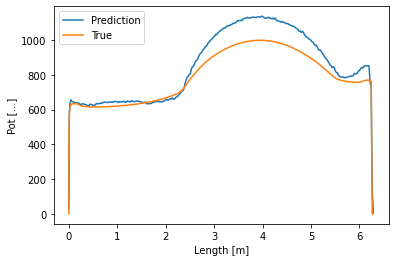
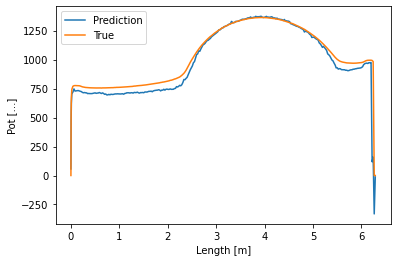
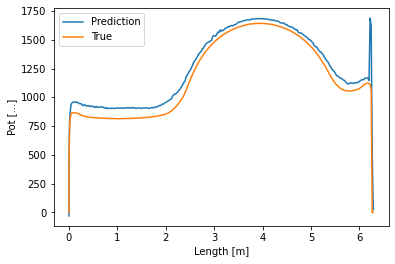
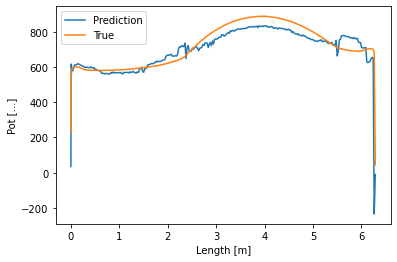
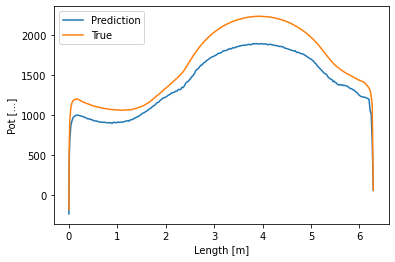
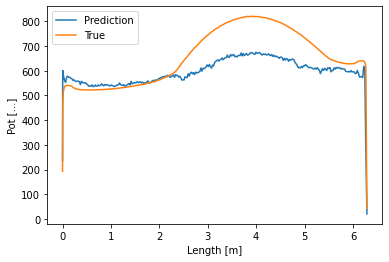
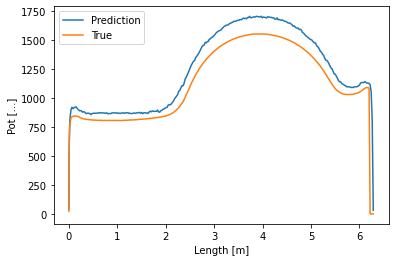
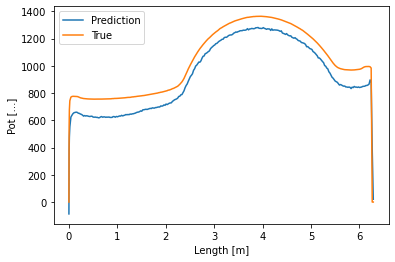
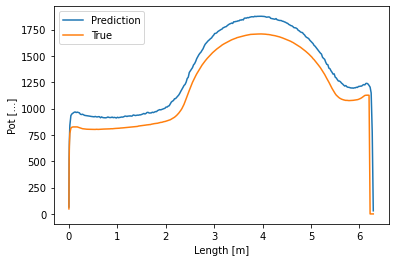
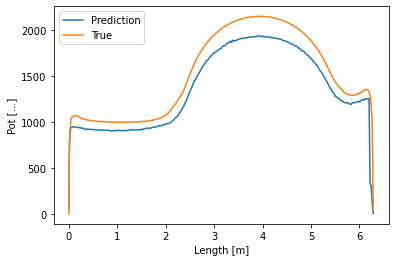
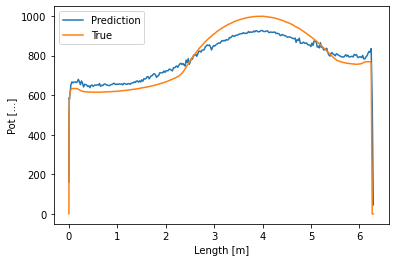
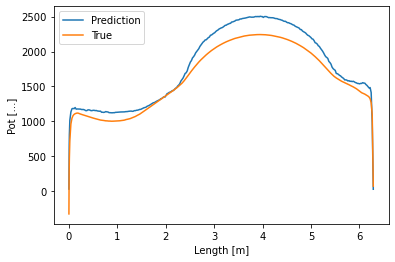
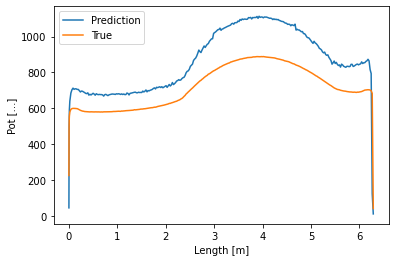
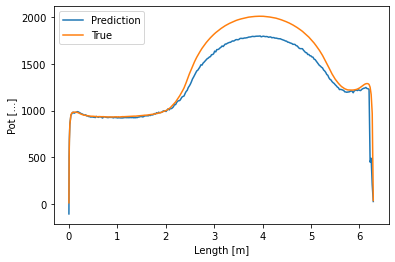
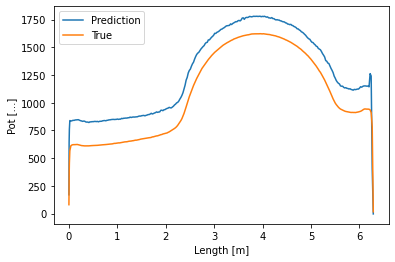
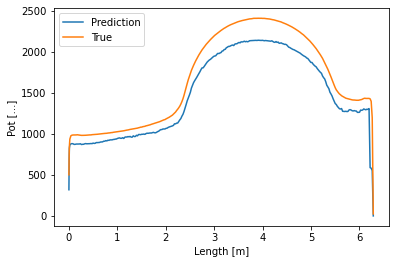
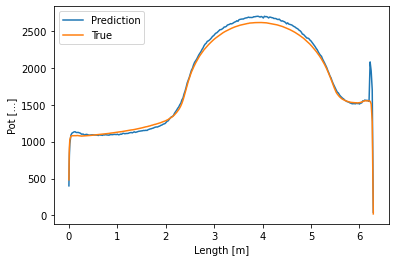
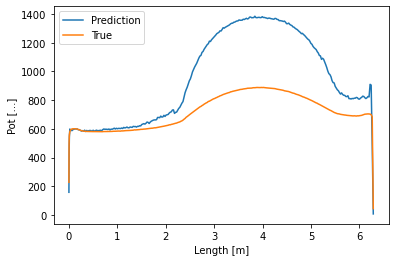
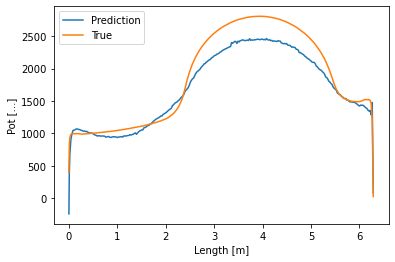
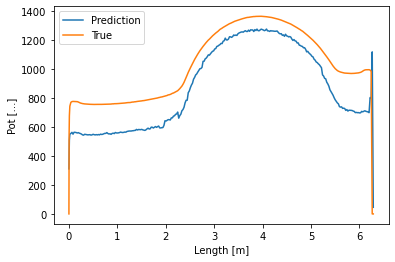
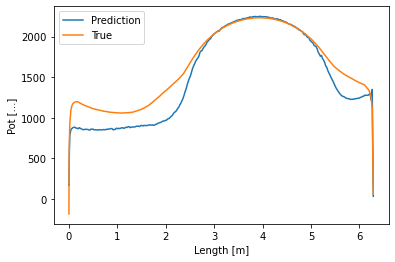
We used the best combination.

**Errors for each set (21)**

7.26, 4.14, 6.8 , 4.93, 13.7 , 9.8 , 9.18, 10.91, 12.02, 9.86, 5.68, 8.81, 20.4 , 5.83, 21.25, 10.67, 2.25, 29.82, 8.95, 17.21, 11.39

**Povprečje vseh 11%**

#### Slike

****

**Drugi parametri**

Bo potrebno določiti najboljše hyperparametre za vsako target funkcijo

ne – napaka 46%

E – napak 233%

Ni – 47%

nn- 140

Te – 18%

Ti – 15,3%

Tn – 5000 %

VE – 21,5 %

Vi – 250 %

Vn – 60%

### Drugi modeli

Linearni model: Navaden in Bayesian

reg = linear\_model.BayesianRidge()

reg = linear\_model.LinearRegression()

**povprečna relativna napaka: 14,6%**