Practical session:

Machine Learning

P. Zurita

2nd European School on the Physics of the EIC and Related Topics
23/06-02/07/2025, Benicàssim, Sapain

Outline

- 😎 Basics of ML
- 😎 What we are going to look at
- 😎 Day 1: regression
- 😎 Day 2: Neural Networks

Basics of (supervised) ML

features: some quantifiable characteristics $\vec{d}$

features: some quantifiable characteristics $\vec{d}$

target: quantity that depends on the features Y

features: some quantifiable characteristics $\vec{d}$

target: quantity that depends on the features Y

target function: the relation between features and targets $f(\vec{d}) = Y$

features: some quantifiable characteristics $\vec{d}$

target: quantity that depends on the features Y

target function: the relation between features and targets $f(\vec{d}) = Y$

X

N values of pressure, temperature and humidity

y

N values of probability of rain

features: some quantifiable characteristics $\vec{d}$

target: quantity that depends on the features Y

target function: the relation between features and targets $f(\vec{d}) = Y$

X → y

N values of pressure, temperature and humidity

$$\hat{f}(X) = \hat{y}$$

N values of probability of rain

features: some quantifiable characteristics $\vec{d}$

target: quantity that depends on the features Y

target function: the relation between features and targets $f(\vec{d}) = Y$

X → y

N values of pressure, temperature and humidity

$$\hat{f}(X) = \hat{y}$$

N values of probability of rain

cost function: “distance” between targets and estimation, to be minimised

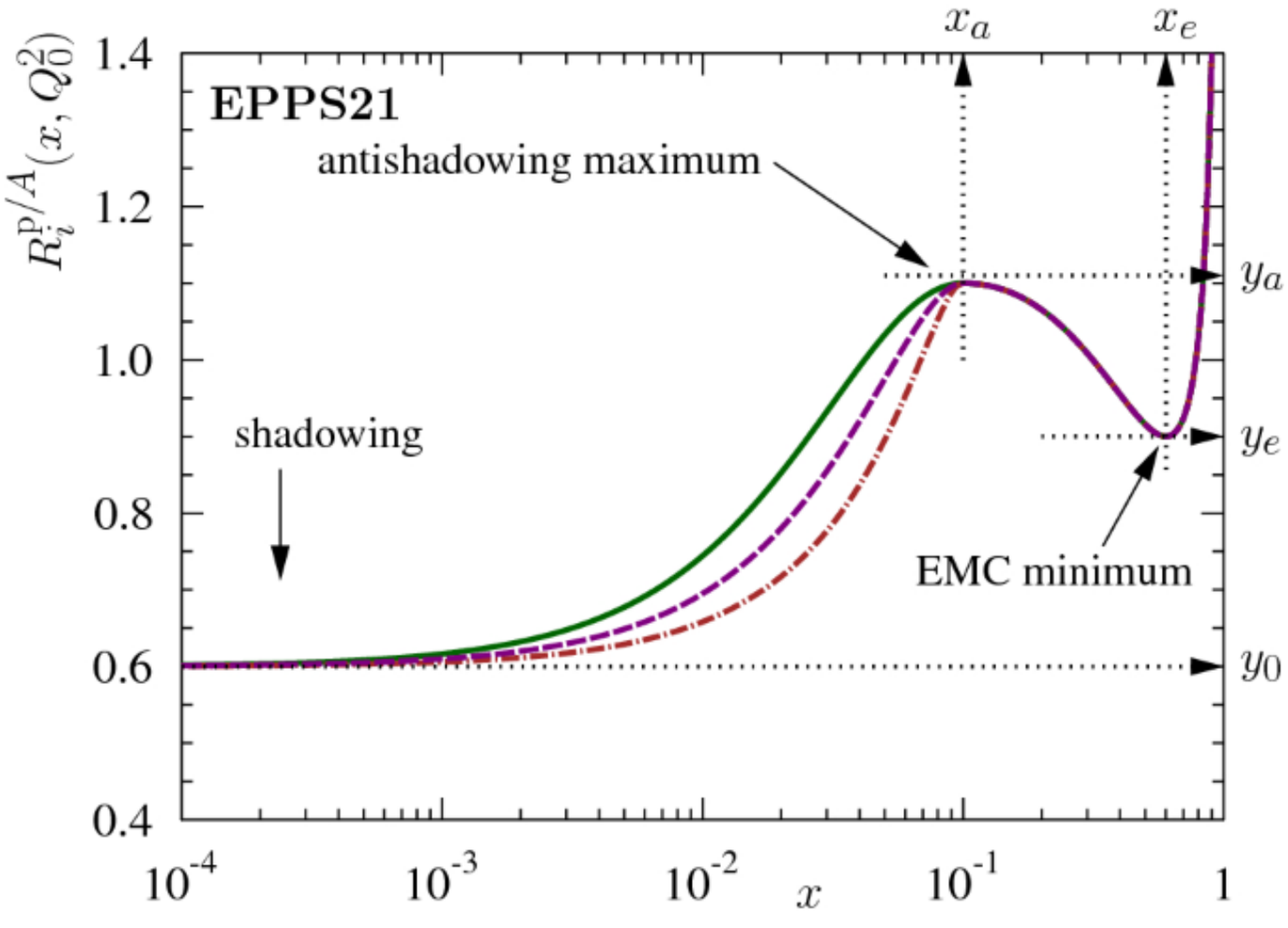
$$C(\hat{y}, y)$$

What we are going to look at

We are going to look different models to study the same problem: the underlying kinematics in $p + A$ collisions.

Drell-Yan in $p + A$ @ $\sqrt{s} = 8.16$ TeV

$$x_{1,2}^{LO} = \frac{M}{\sqrt{s}} e^{\pm \text{rapidity}}$$



$$x_2^{LO} = \frac{M}{\sqrt{s}} e^{-\text{rapidity}}$$

$$x_2^{NLO} = ?$$

rapidity $\in [-2.5, 2.5]$

$M \in [10\ \text{GeV},\ 600\ \text{GeV}]$

$N_{data} \sim 10^5$

https://scikit-learn.org/stable/

Let’s start!

First: let’s take a look at the data for inspiration

Normally at this point one looks for “gaps” (missing entries) and applies some fix to that.

It is also common to apply preprocessing *feature scaling*:

min-max scaling $\frac{x_{j,i} - x_{j,min}}{x_{j,max} - x_{j,min}} \in [0,1]$ MinMaxScaler(feature_range)

standardisation $\frac{x_{j,i} - \mu_j}{\sigma_j} \sim N(0,1)$ StandardScaler

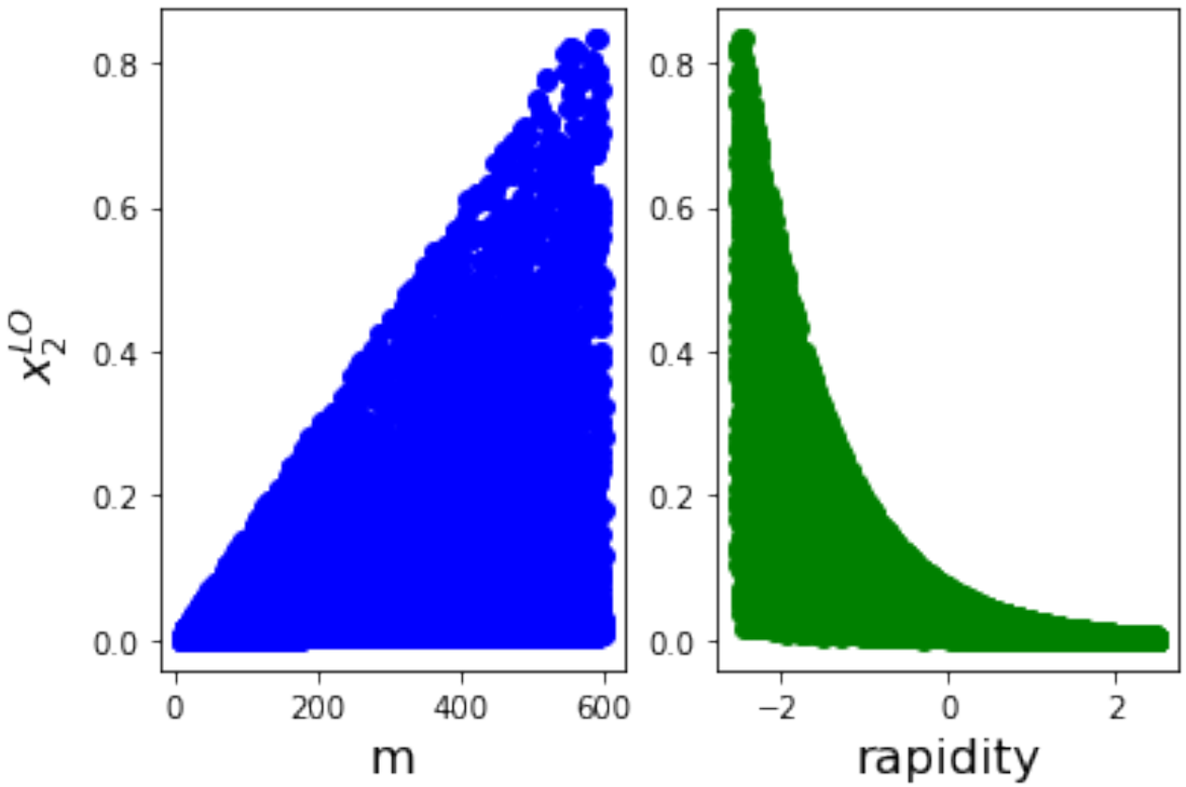
First: let’s take a look at the data for inspiration

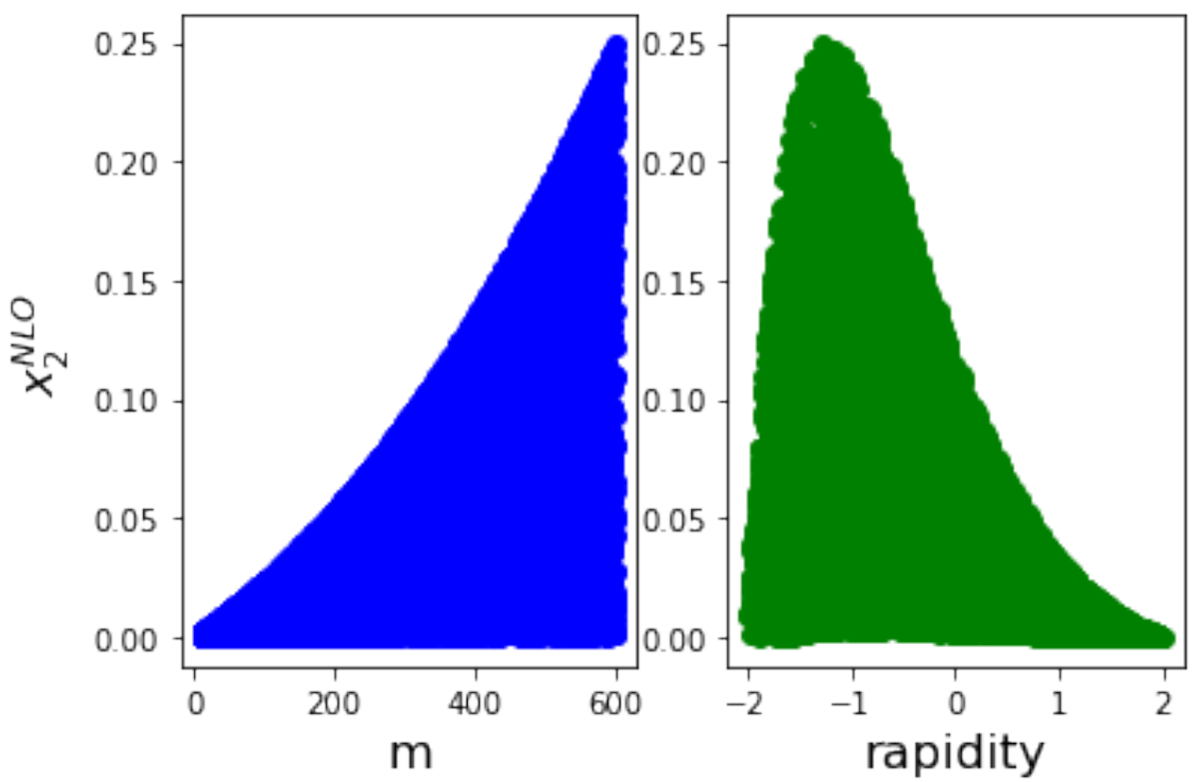
Our data will not have uncertainties. Everything is perfect.

data = 🌈🦄☀️🦋🌻

First: let’s take a look at the data for inspiration

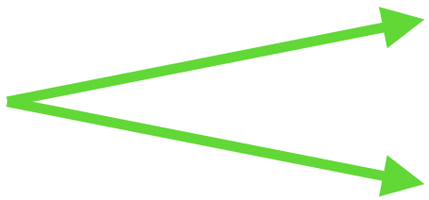
Open EIC_data.ipynb





I feel throughly uninspired.

X

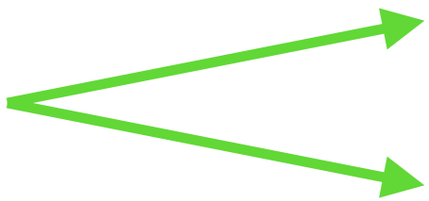


training set X_{train}

test set X_{test}

data set
many points
one or more features

y



training target y_{train}

test target y_{test}

data set
many points

This is very important to avoid overfitting

One can split as one prefers

Day 1:

regression

Regression:

Linear model

$$\hat{f}(X) = \hat{y} = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n = \vec{\theta} \cdot \vec{x}$$

$$MSE = \frac{1}{N_{data}} \Sigma_{i=1}^{N_{data}} \left(\vec{\theta} \cdot \vec{x}^{(i)} - y^i \right)^2$$

Open EIC_LO_LinearModel.ipynb

$$\hat{f}(M, \text{rapidity}) = a_1 M + a_2 \text{rapidity}$$

Open EIC_LO_LinearModel.ipynb

```
def training (X_train, y_train, X_test, y_test, filename, model):

    # train
    model.fit(X_train, y_train)

    # save trained model
    joblib.dump(model, filename)

    # load model
    loaded_model = joblib.load(filename)

    # R^2
    R2   = loaded_model.score(X_test, y_test)

    # predicted targets
    y_fit = loaded_model.predict(X_test)
    # estimated targets
    y_est = loaded_model.predict(X_train)


    return R2, y_fit, y_est, model.coef_
```

Open EIC_LO_LinearModel.ipynb

```
def training (X_train, y_train, X_test, y_test, filename, model):

    # train
    model.fit(X_train, y_train)

    # save trained model
    joblib.dump(model, filename)

    # load model
    loaded_model = joblib.load(filename)

    # R^2
    R2   = loaded_model.score(X_test, y_test)

    # predicted targets
    y_fit = loaded_model.predict(X_test)
    # estimated targets
    y_est = loaded_model.predict(X_train)


    return R2, y_fit, y_est, model.coef_
```

Open EIC_LO_LinearModel.ipynb

```
def training (X_train, y_train, X_test, y_test, filename, model):

    # train
    model.fit(X_train, y_train)

    # save trained model
    joblib.dump(model, filename)

    # load model
    loaded_model = joblib.load(filename)

    # R^2
    R2   = loaded_model.score(X_test, y_test)

    # predicted targets
    y_fit = loaded_model.predict(X_test)
    # estimated targets
    y_est = loaded_model.predict(X_train)


    return R2, y_fit, y_est, model.coef_
```

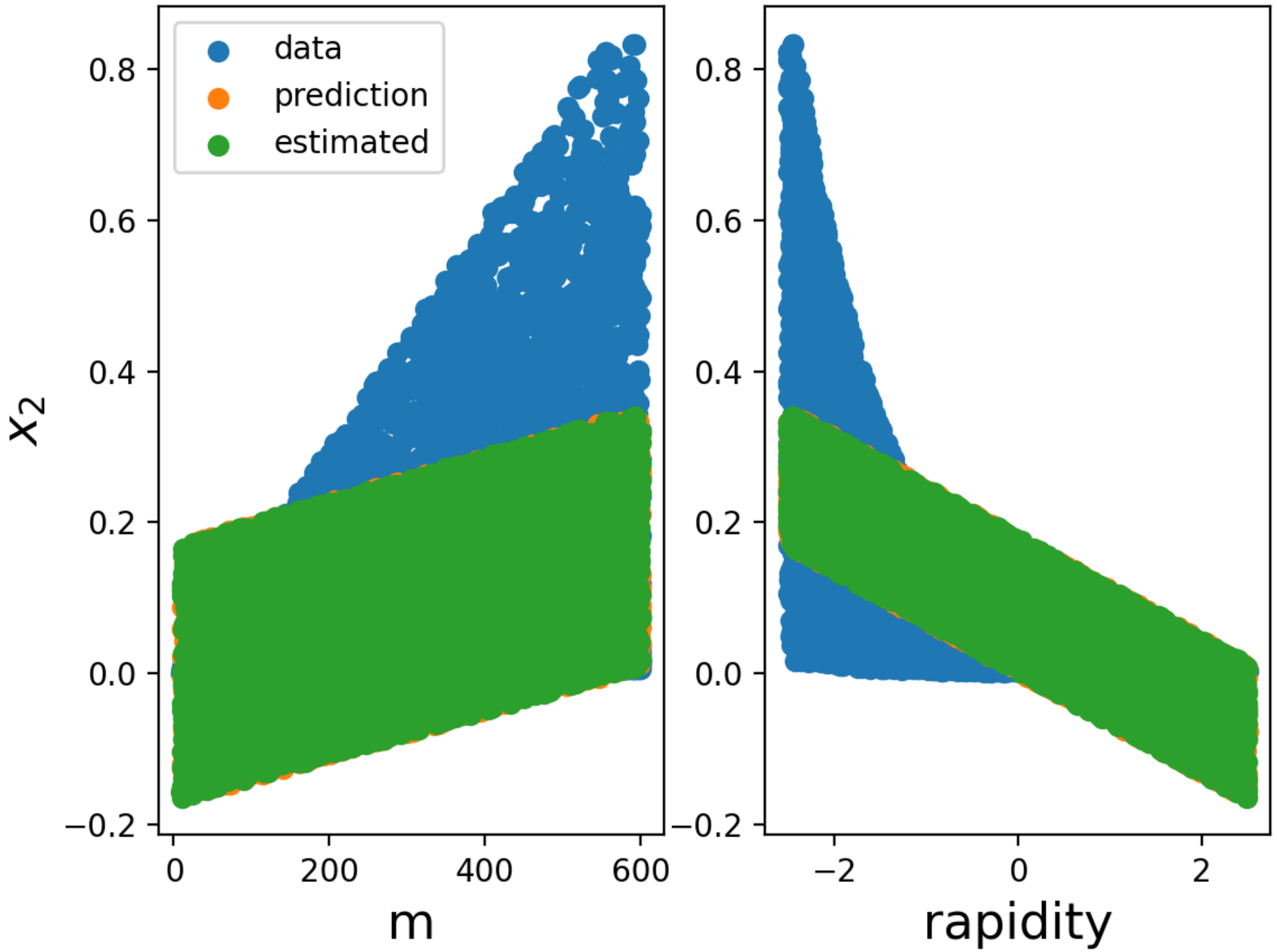
Open EIC_LO_LinearModel.ipynb

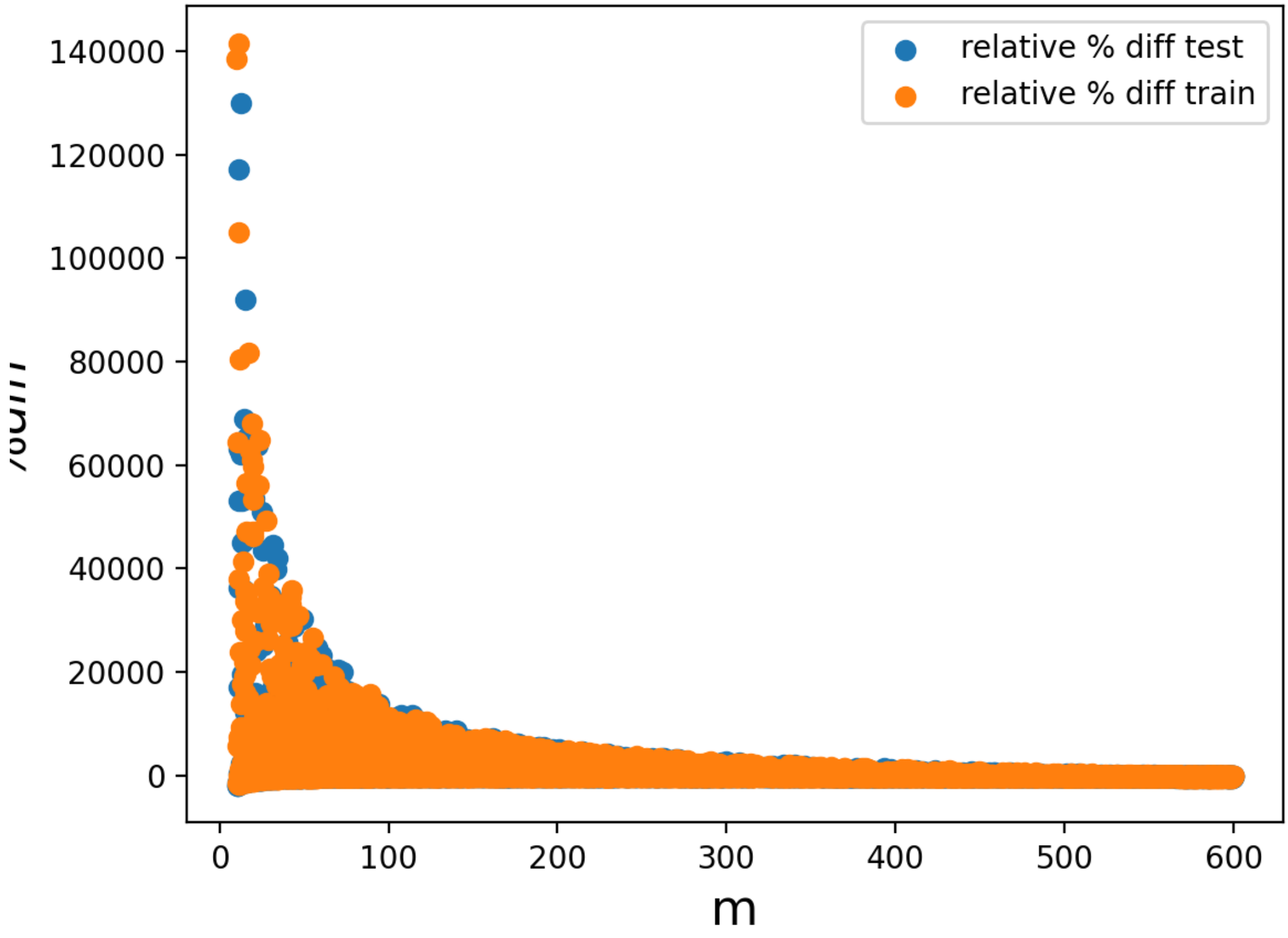
```
def training (X_train, y_train, X_test, y_test, filename, model):

    # train
    model.fit(X_train, y_train)

    # save trained model
    joblib.dump(model, filename)

    # load model
    loaded_model = joblib.load(filename)

    # R^2
    R2   = loaded_model.score(X_test, y_test)

    # predicted targets
    y_fit = loaded_model.predict(X_test)
    # estimated targets
    y_est = loaded_model.predict(X_train)


    return R2, y_fit, y_est, model.coef_
```

Open EIC_LO_LinearModel.ipynb

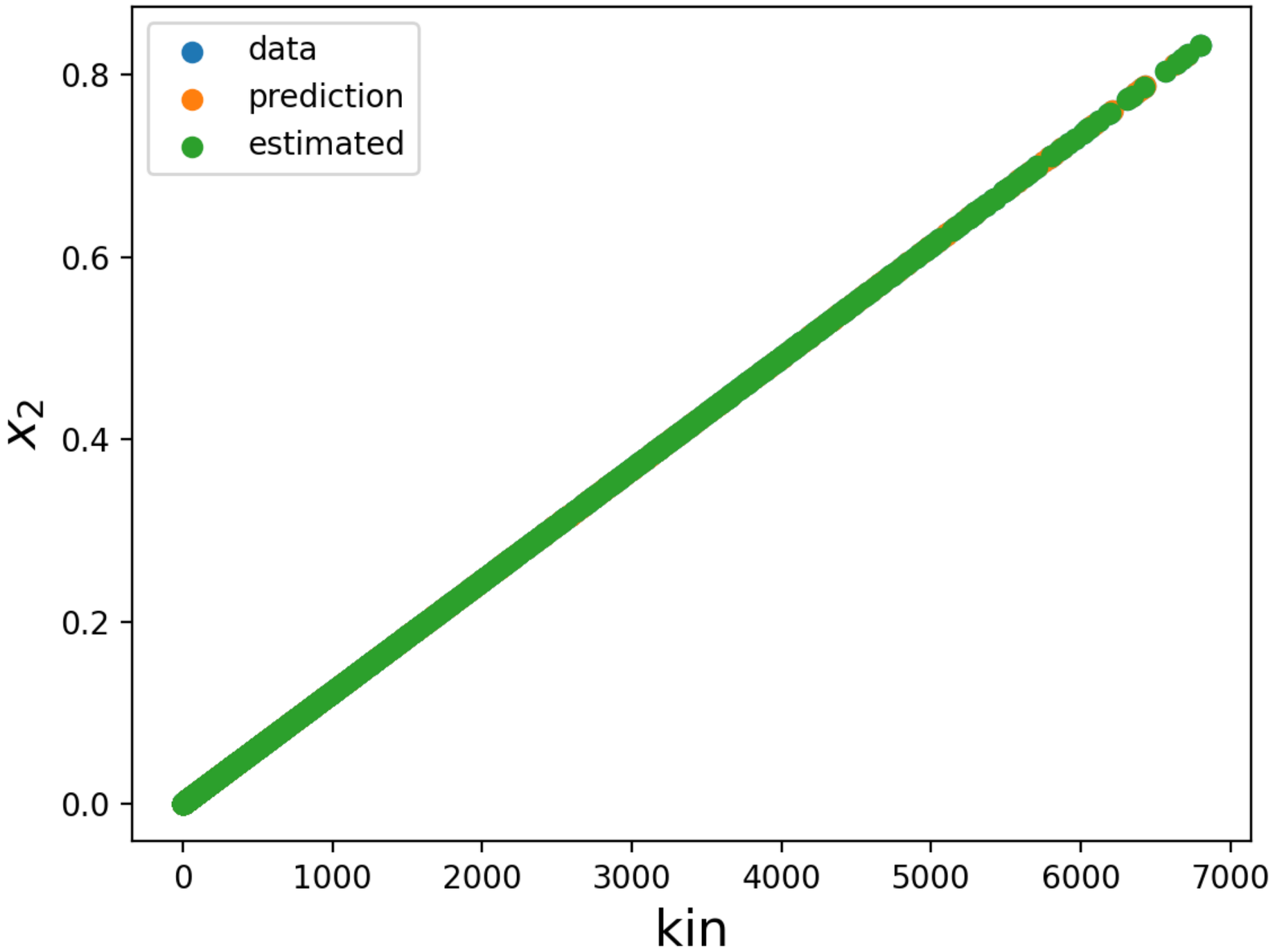
```
def training (X_train, y_train, X_test, y_test, filename, model):

    # train
    model.fit(X_train, y_train)

    # save trained model
    joblib.dump(model, filename)

    # load model
    loaded_model = joblib.load(filename)

    # R^2
    R2   = loaded_model.score(X_test, y_test)

    # predicted targets
    y_fit = loaded_model.predict(X_test)
    # estimated targets
    y_est = loaded_model.predict(X_train)


    return R2, y_fit, y_est, model.coef_
```

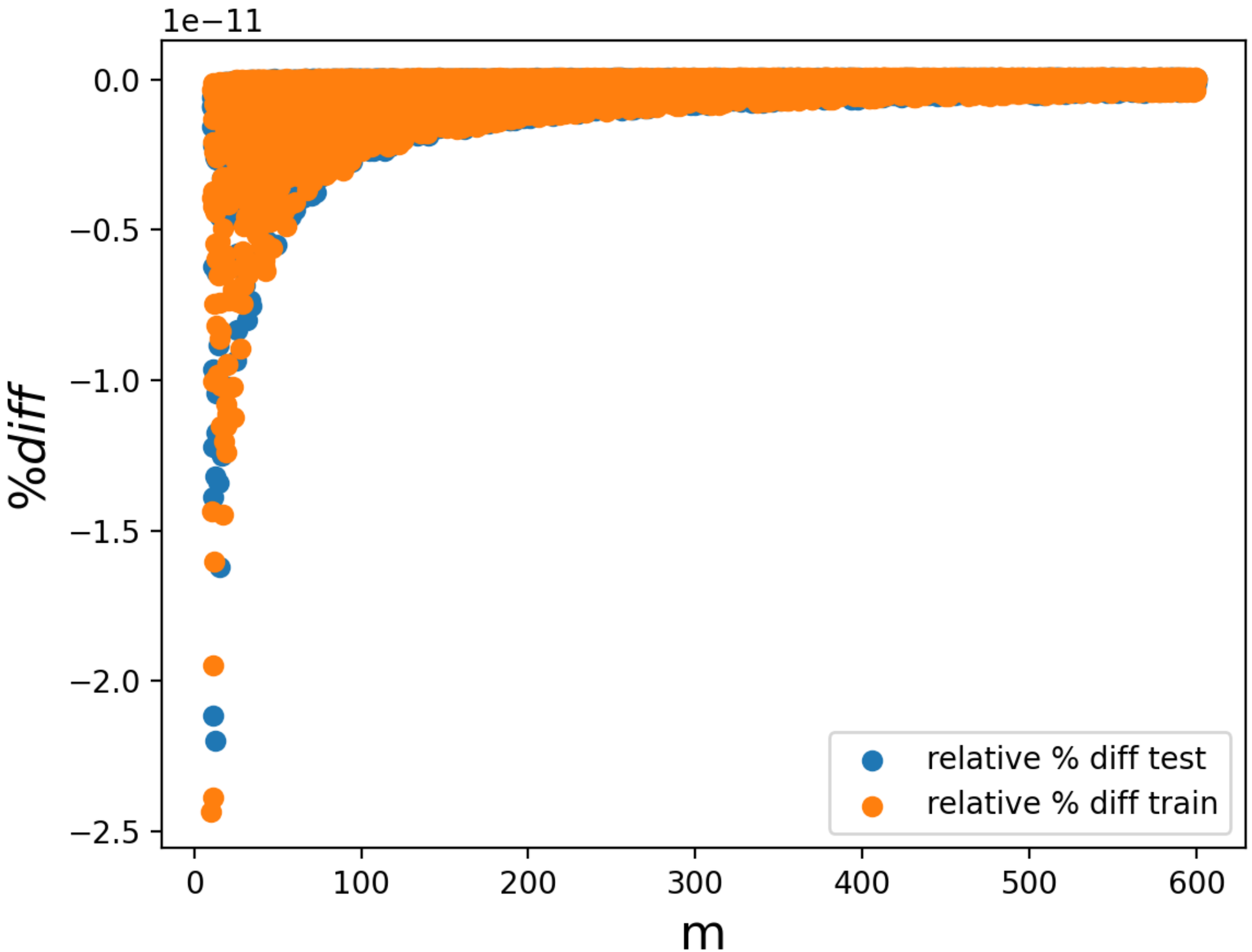
Brute force attempt




relative % diff test
relative % diff train
140000
120000
100000
80000
60000
40000
20000
0
0
100
200
300
400
500
600
m

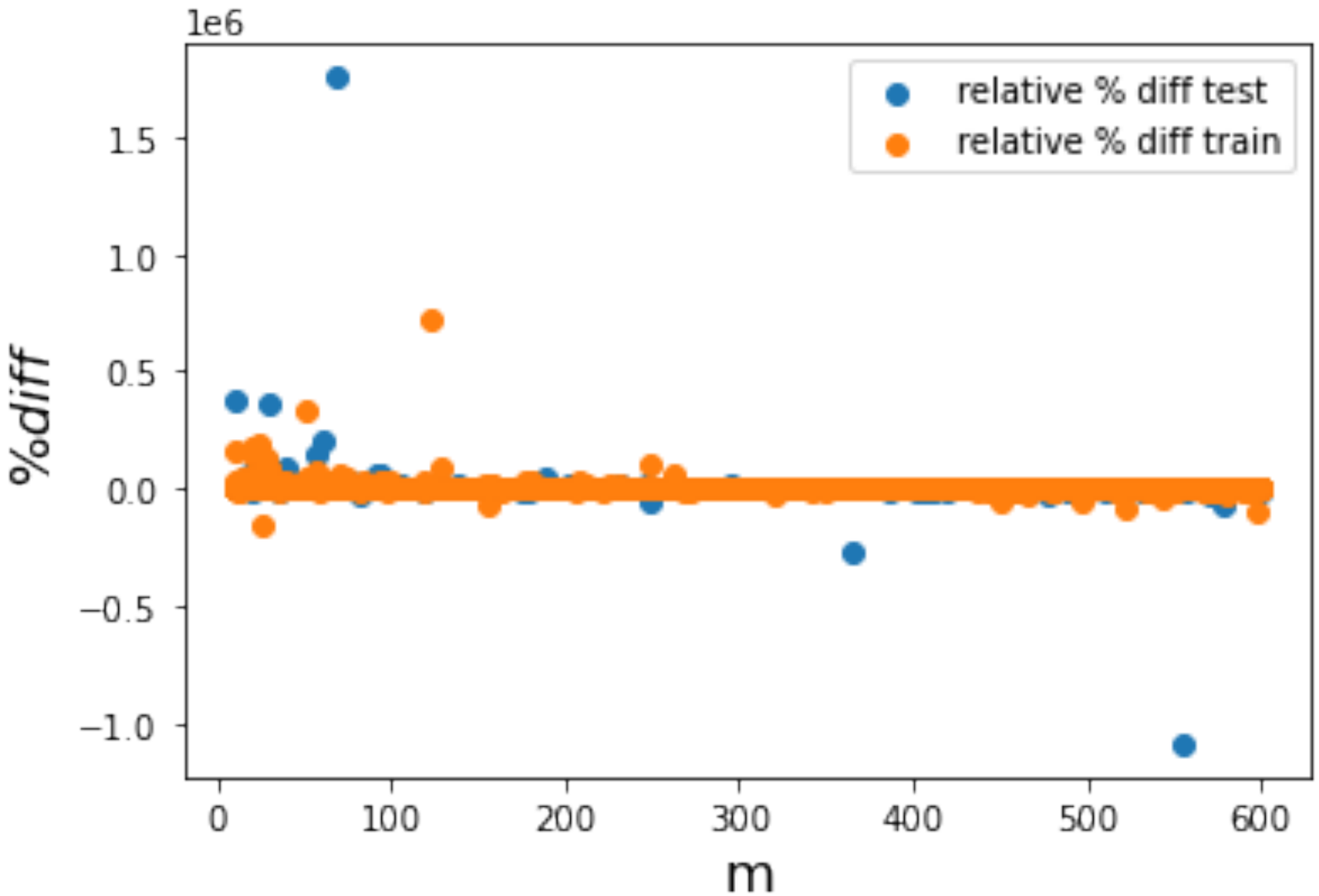
Smarter attempt: build feature Me^{-y} and use that as X



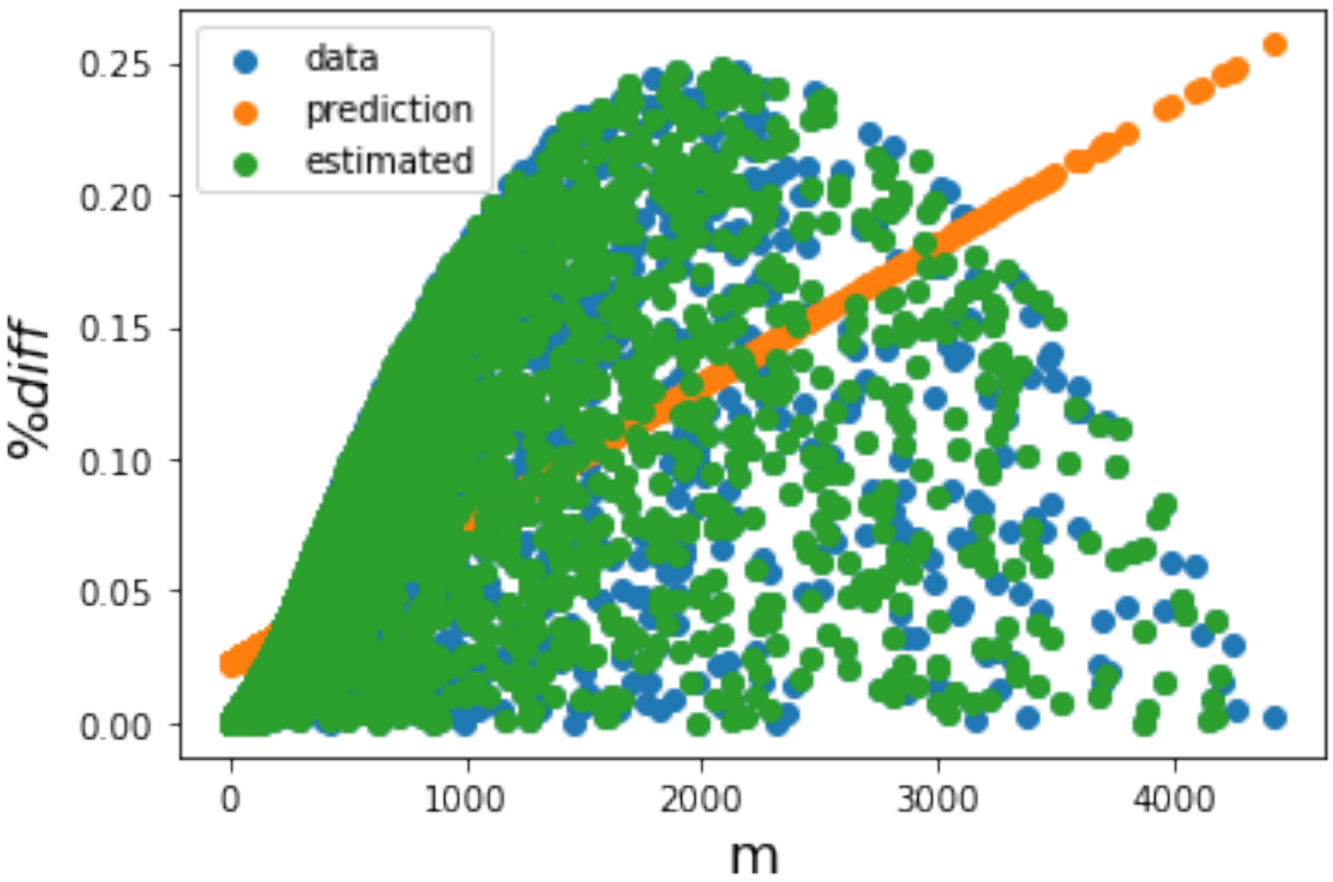

1e−11
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
%diff
0
100
200
300
400
500
600
m
relative % diff test
relative % diff train

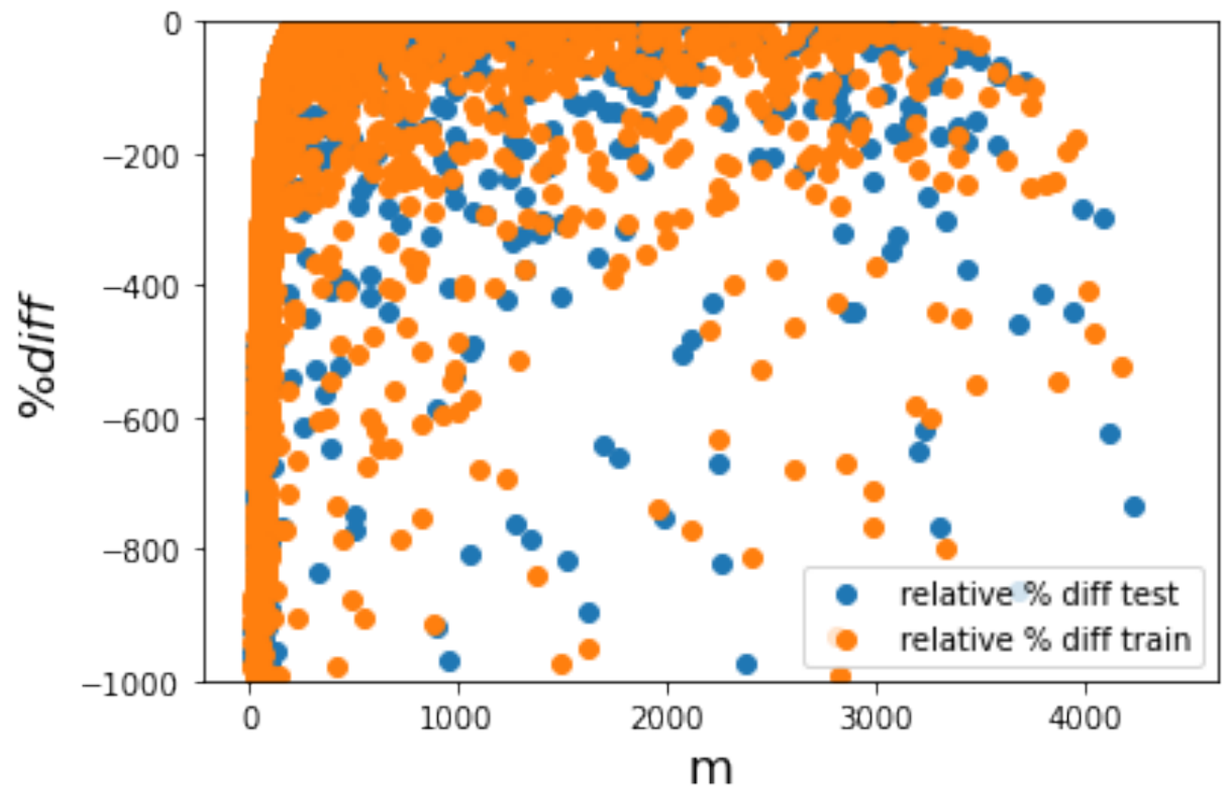
Let’s go to NLO! Open EIC_NLO_LinearModel.ipynb

Brute force attempt

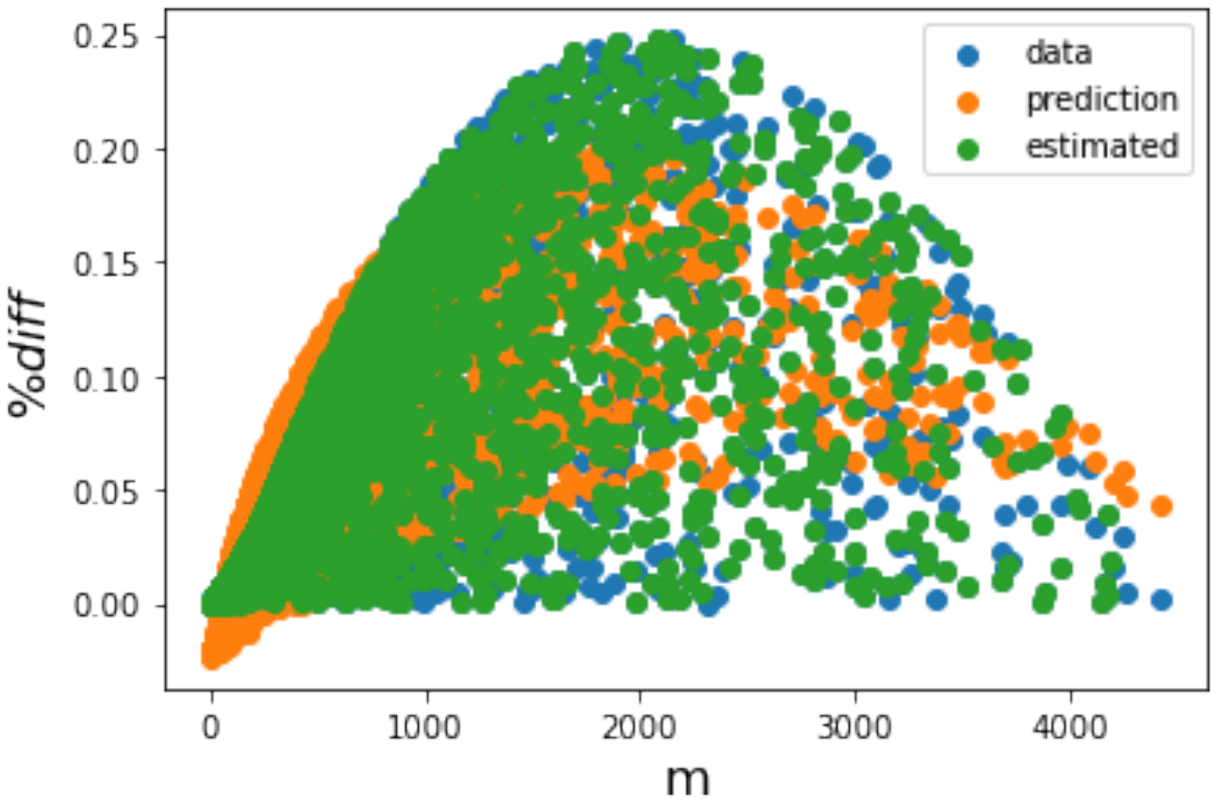


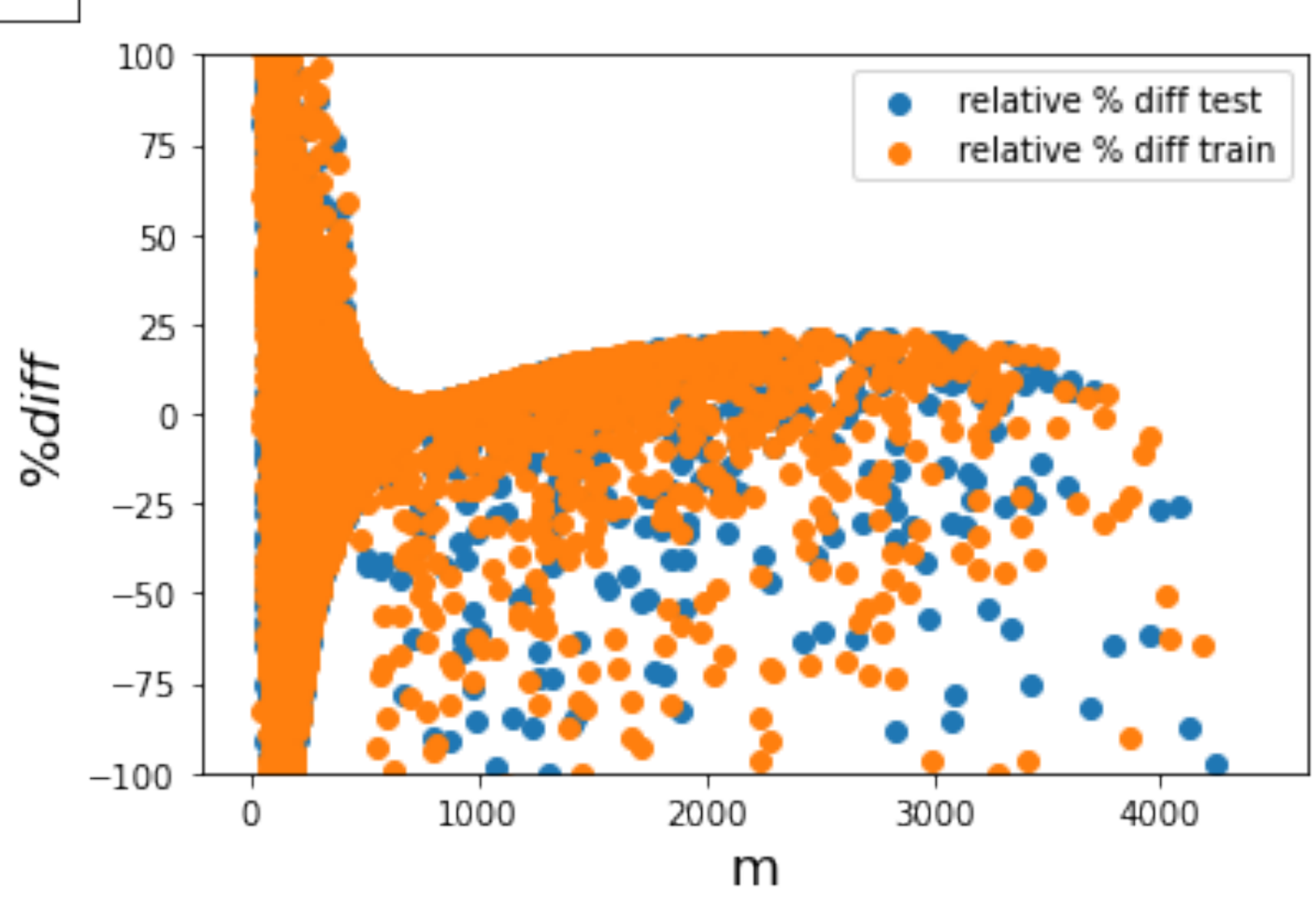
Smarter attempt1: build feature Me^{-y} and use that as X



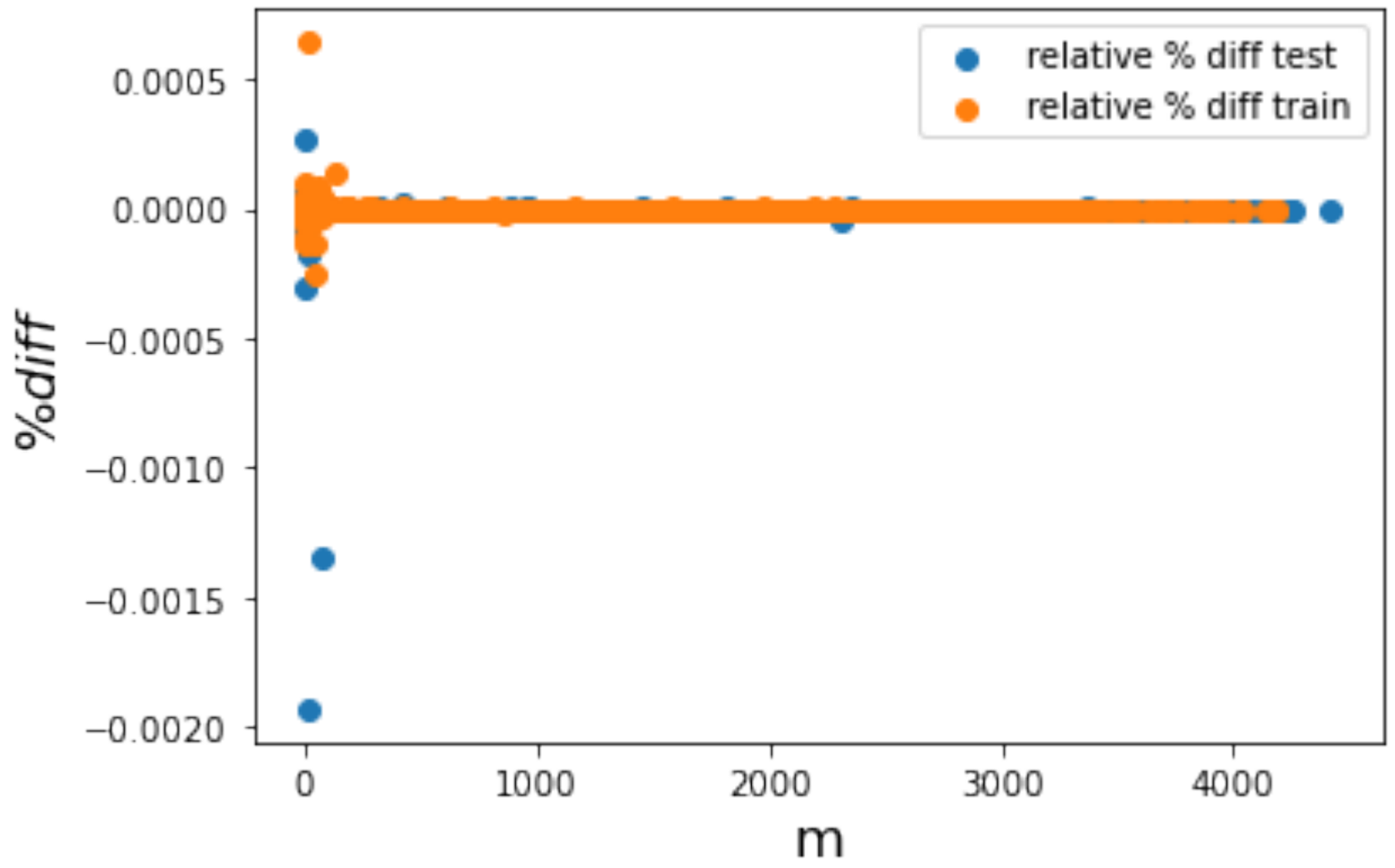


Smarter attempt2: build 2 features Me^{-y} and yMe^{-y}





The actual solution:



I’ll buy ice-cream to the first person (non lecturer!) to find the correct basis and coefficients before Sunday.

Regression: the kernel trick

Kernel trick: introducing non-linearity to linear techniques

Let's say we have a function F that maps features into (better) features. When minimising, the cost function depends on (Euclidean) distance (= dot product).

We define the kernel $K(x_i, x_j)$

$$K(x_i, x_j) = F(x_i) \cdot F(x_j)$$

The kernel stores everything, we do not need to know F.

$$K_{RBF}(x_i, x_j) = \exp\left(-\frac{||x_i - x_j||^2}{2l^2} \right)$$

l = length scale parameter

Cross-validation

How to make sure that we are not just being lucky?

Repeat the training-testing procedure with different partitions of the data

Day 2:

Neural Networks

Gradient Descent

We are going to start by a learning algorithm that is incredibly useful, and not only used by NN:

Gradient descent

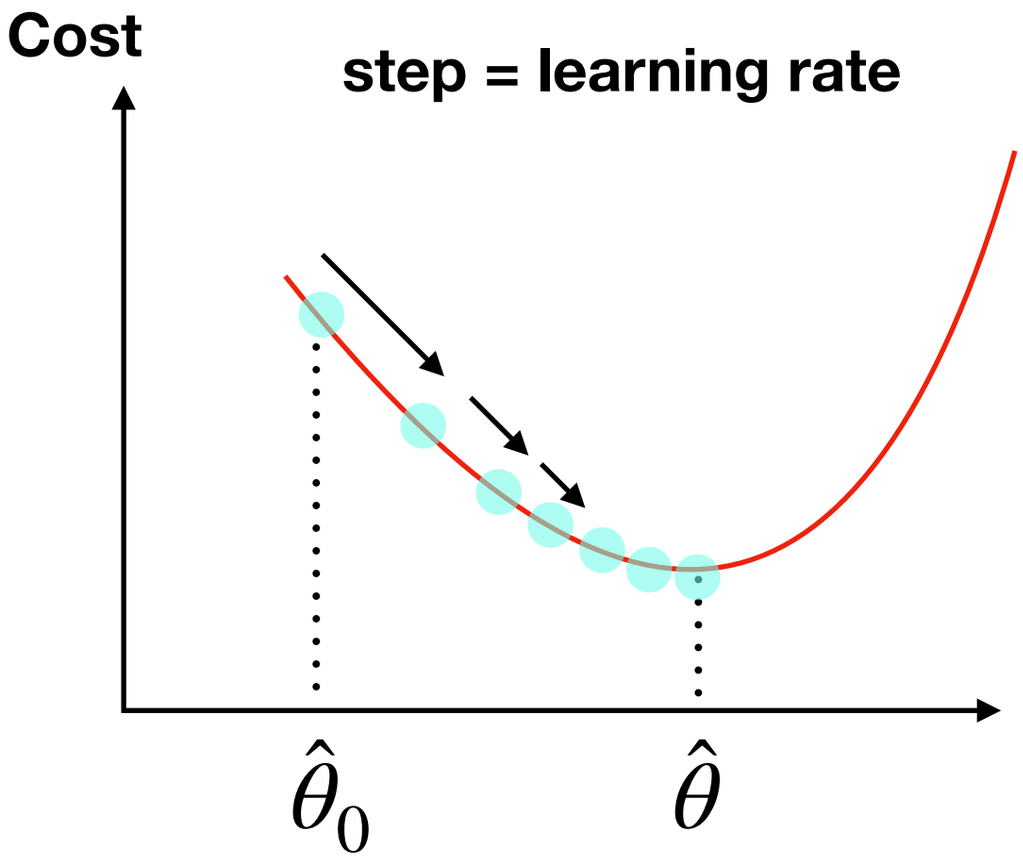
$\vec{\theta}$: parameters to learn

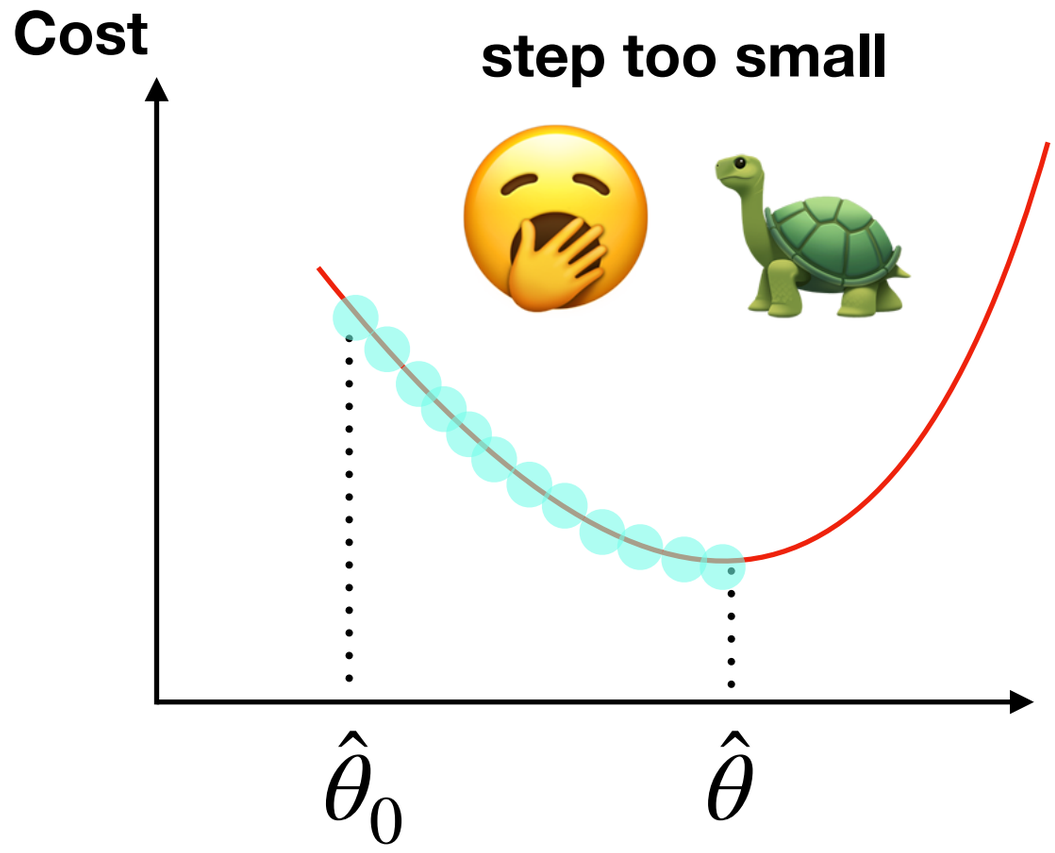
Step 1: give random values to θ (random initialisation)

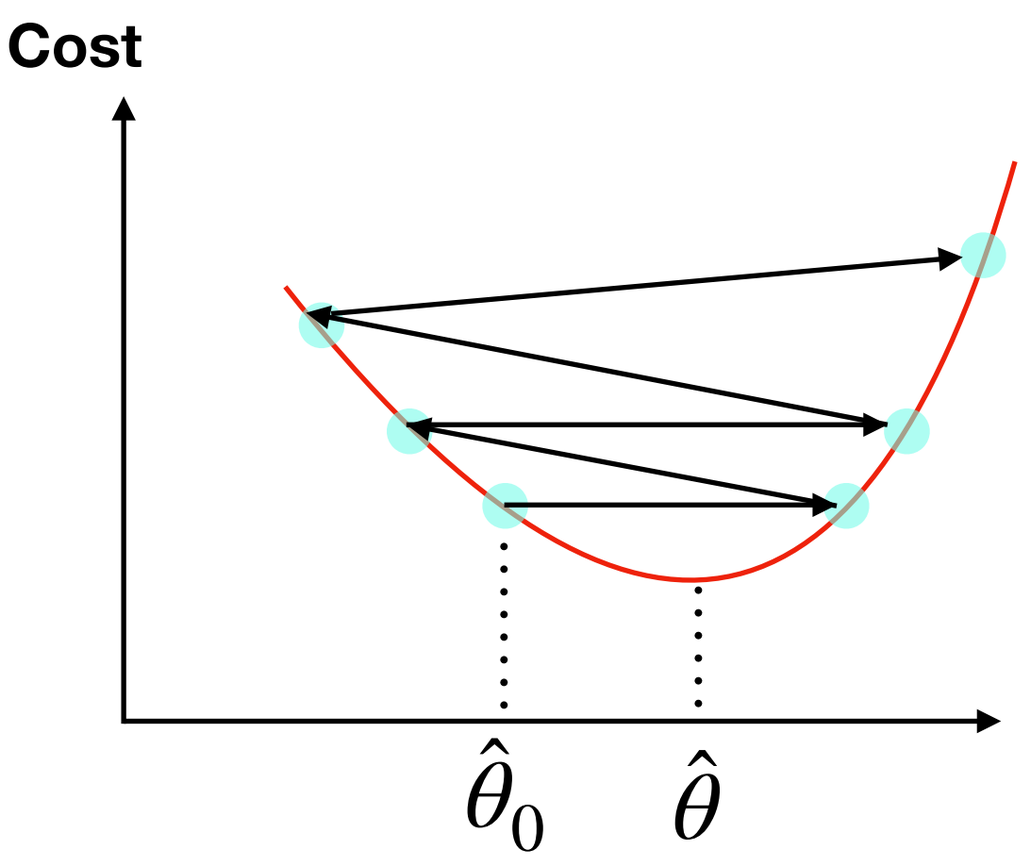
Step 2: compute locally the gradient of the cost function

Step 3: take a step in the direction of maximum decrease of the gradient

Step 4: repeat from Step 2







step too large

- might end up in local minimum
- might take forever
- might never converge to anything if one stops it too early

$$MSE = \frac{1}{N_{data}} \Sigma_{i=1}^{N_{data}} \left(\vec{\theta} \cdot \vec{x}^{(i)} - y^{i} \right)^2$$

$$\frac{\partial}{\partial \theta_j} MSE = \frac{2}{N_{data}} \Sigma_{i=1}^{N_{data}} \left(\vec{\theta} \cdot \vec{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

$$\nabla_\theta MSE = \frac{2}{N_{data}} X^T (X\theta - y)$$

$$\vec{\theta}_{new} = \vec{\theta}_{old} - \text{learning rate} * \nabla_\theta MSE$$

$$MSE = \frac{1}{N_{data}} \Sigma_{i=1}^{N_{data}} \left(\vec{\theta} \cdot \vec{x}^{(i)} - y^{i} \right)^2$$

$$\frac{\partial}{\partial \theta_j} MSE = \frac{2}{N_{data}} \Sigma_{i=1}^{N_{data}} \left(\vec{\theta} \cdot \vec{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

$$\nabla_\theta MSE = \frac{2}{N_{data}} X^T (X\theta - y)$$

This is the Batch Gradient Descent algorithm. It is terribly slow, so we use **Stochastic Gradient Descent**

Stochastic Gradient Descent

🙃 faster than BGD

🙃 good for huge training sets

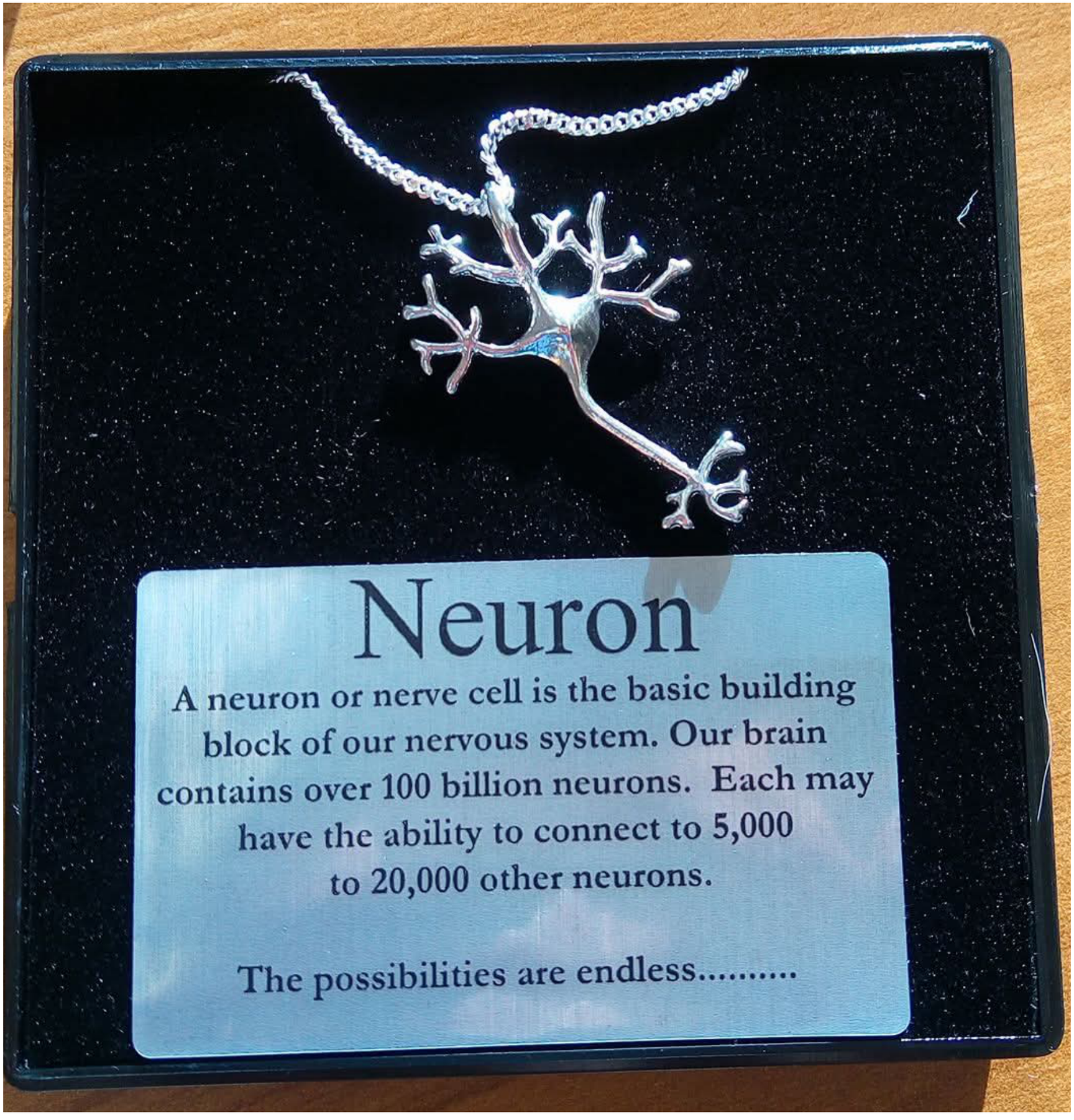
☹️ less regular than BGD

☹️ never settles down

😐 randomness

EIC_NLO_LinearModel.ipynb

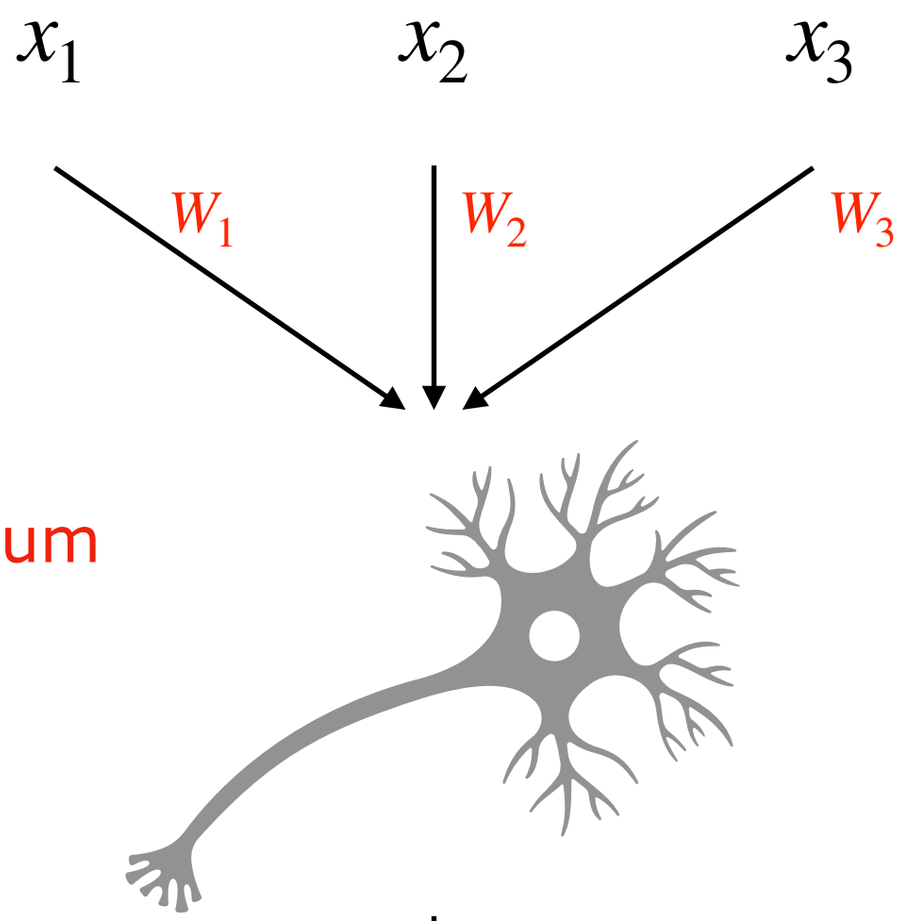
Neural Networks


Neuron
A neuron or nerve cell is the basic building block of our nervous system. Our brain contains over 100 billion neurons. Each may have the ability to connect to 5,000 to 20,000 other neurons.
The possibilities are endless.....

The Perceptron (1957!)

inputs

weights



takes in the weighted sum

$$z = x^T W$$

applies a step function

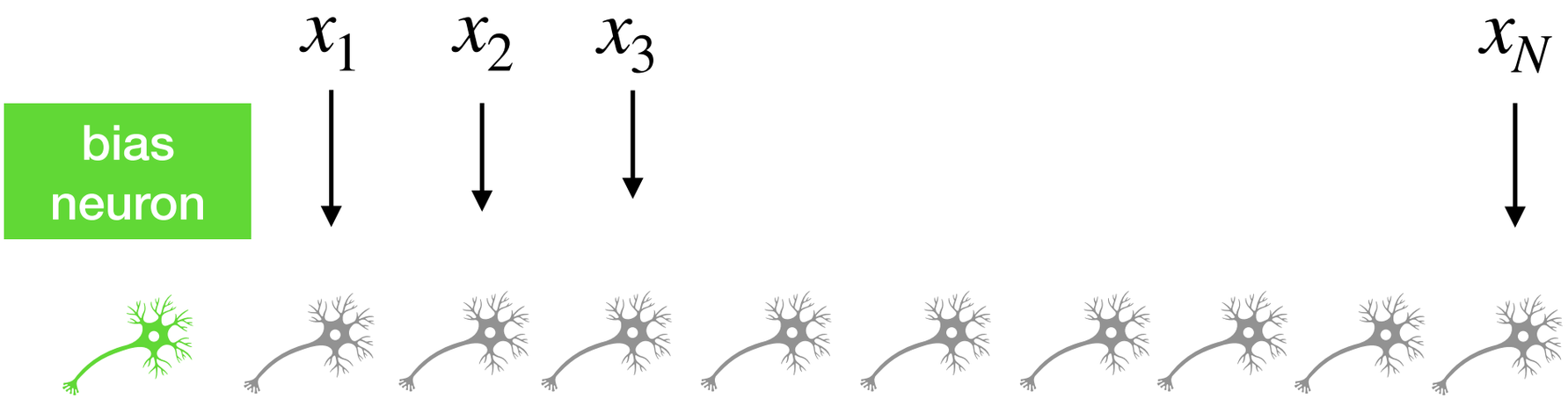
step(z)

step(z)

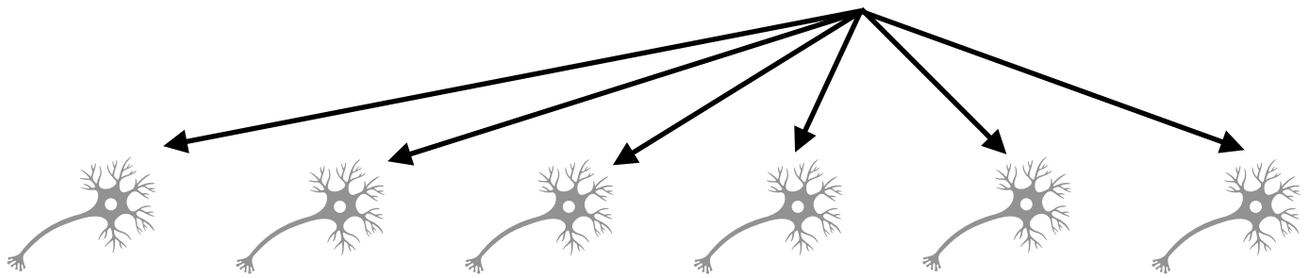
output

this is the **threshold logic unit** (TLU) or **linear threshold unit** (LTU)

For the Perceptron, one stacks together several TLUs/LTUs



fully democratic connections



output layer
(fully connected)

outputs

$$h_{W,b}(X) = \phi(XW + b)$$

$$h_{W,b}(X) = \phi(XW + b)$$

X is the matrix of input features (1 row/instance, 1 column/feature)

W is a matrix with all the weights except the bias neuron (1 row/input neuron, 1 column/neuron in the layer)

b is the vector of bias weights (1 column/neuron in the layer)

ϕ is the **activation function** (not necessarily a step function)

$$w_{i,j}^{new} = w_{i,j}^{old} + \text{learning rate} * (y_j - \hat{y}_j) x_i$$

$w_{i,j}$: weight from ith input neuron to jth output neuron

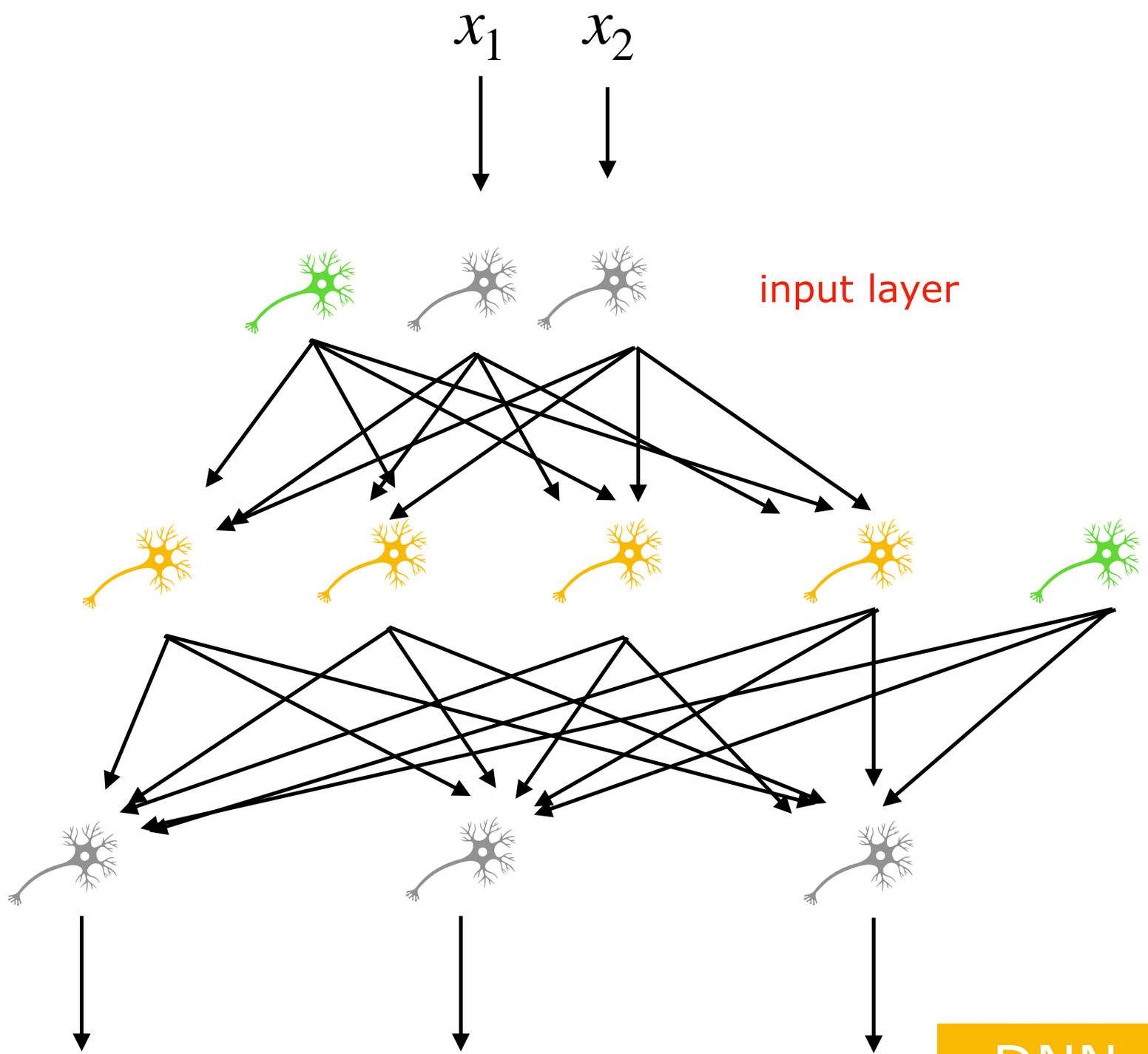
x_i : ith input value for the current training instance

$\hat{y}_j$: output of the jth output neuron for the current training instance

y_j : target output of the jth output neuron for the current training instance

Perceptron
=
SGDClassifier(loss="perceptron",learning_rate="constant,
eta0=1,penalty=None)

Multilayer Perceptron (MLP)



DNN="many" hidden layers

How to train your network?

Using **back propagation**

=

automatised **Gradient Descent**

1. take a small number of training instances at a time (mini-batch) and go over the full training set multiple times. Each pass is an “epoch”.

2. forward pass for each mini-batch:

pass it to the input layer (into the first hidden layer)

compute the output of the first hidden layer

pass them to the second hidden layer as input

…

get final output

3. compute the error of the network's output

4. compute how much each connection contributed to the error using the **chain rule**

5. compute how much of the error contributions came from each connection in the layer below. Repeat until the input layer is reached (**back**)

6. update all weights using **Gradient Descent**

It is key here to use activation functions that allow for a gradient to be computed

Popular choices for activation functions

sigmoid $$\sigma(z) = \frac{1}{1 + e^{-z}}$$

hyperbolic tangent $$\tanh(z) = 2\sigma(2z) - 1$$

Rectified Linear Unit $$\text{ReLU}(z) = \max(0,z)$$

The activation function is key to incorporate non-linearities to the training.

MLPRegressor

hidden_layer_sizes=*(100,)*

activation=*'identity', 'logistic', 'tanh', 'relu'*

solver=*'lbfgs', 'sgd', 'adam'*

learning_rate=*'constant', 'invscaling', 'adaptive'*

learning_rate_init=*0.001*

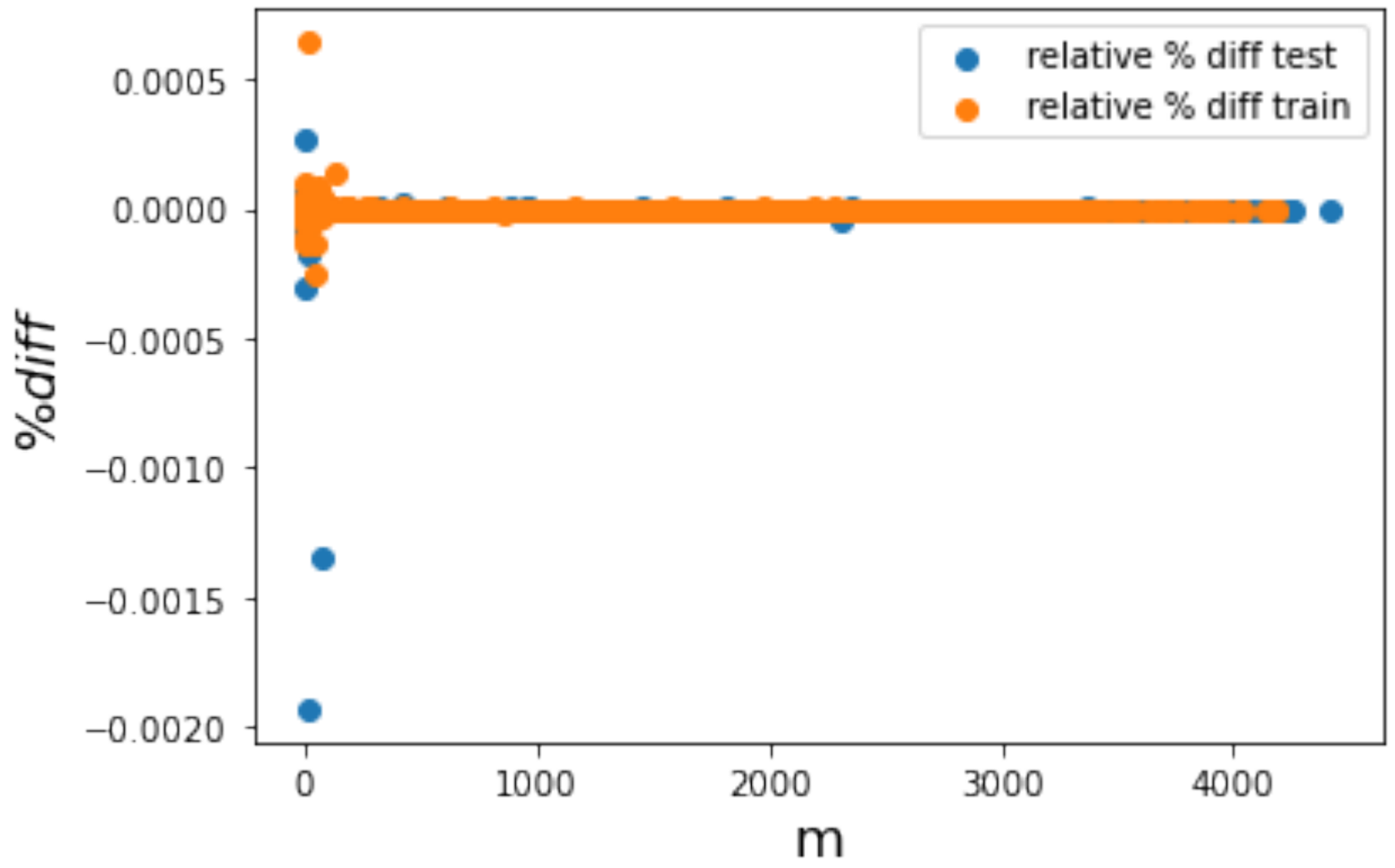
power_t=*0.5*

max_iter=*200*

shuffle=*True*

tol=*0.0001*

Remember the real solution:



I'll buy ice-cream to the first person/team (non lecturer!) to find a NN with this good of a result before Sunday.