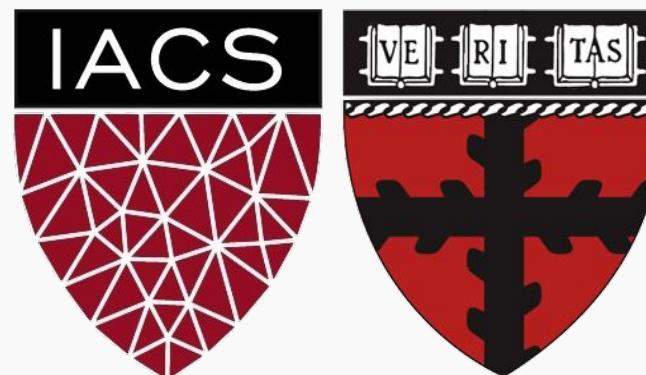


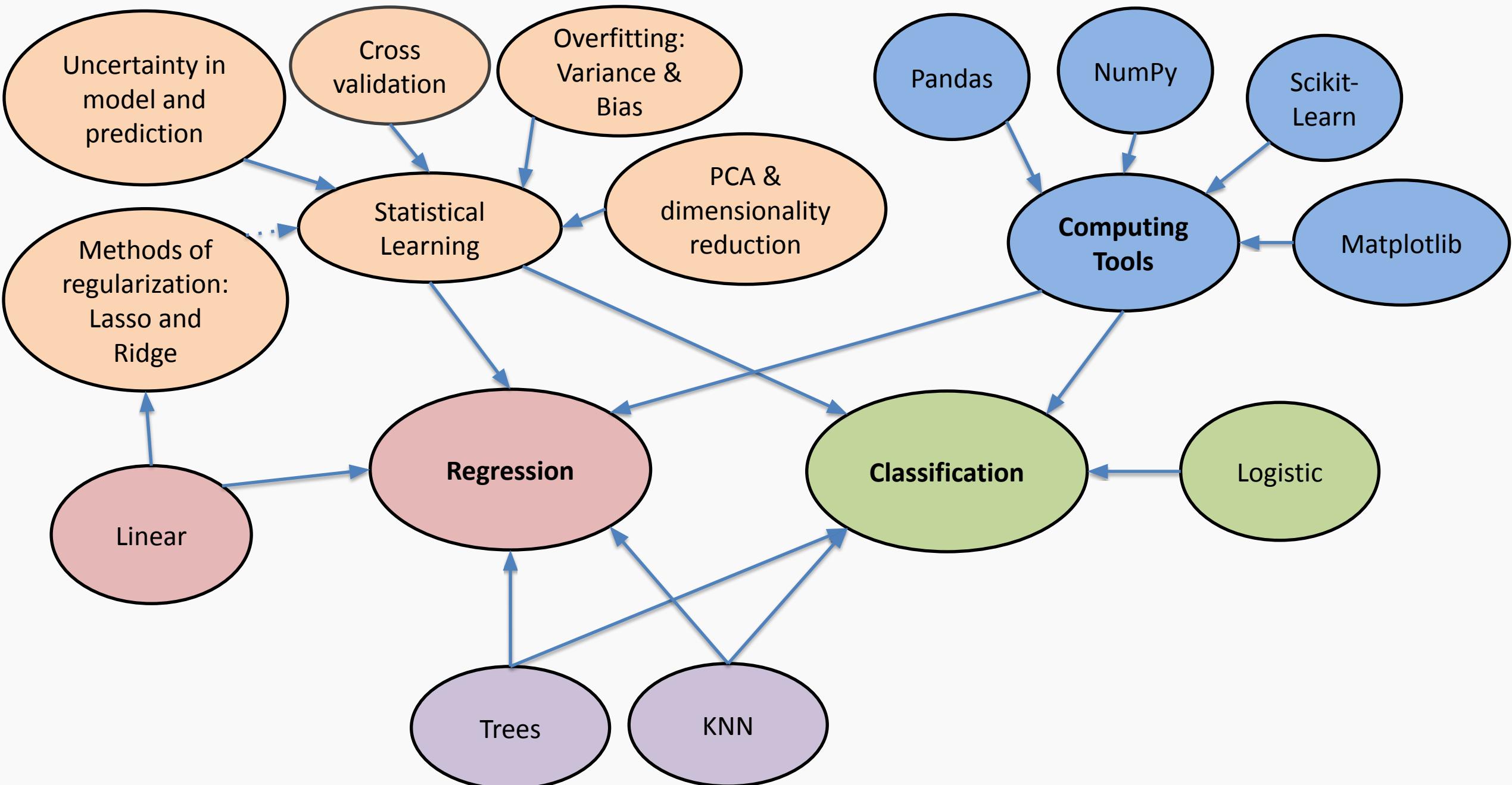
Lecture 24: Review

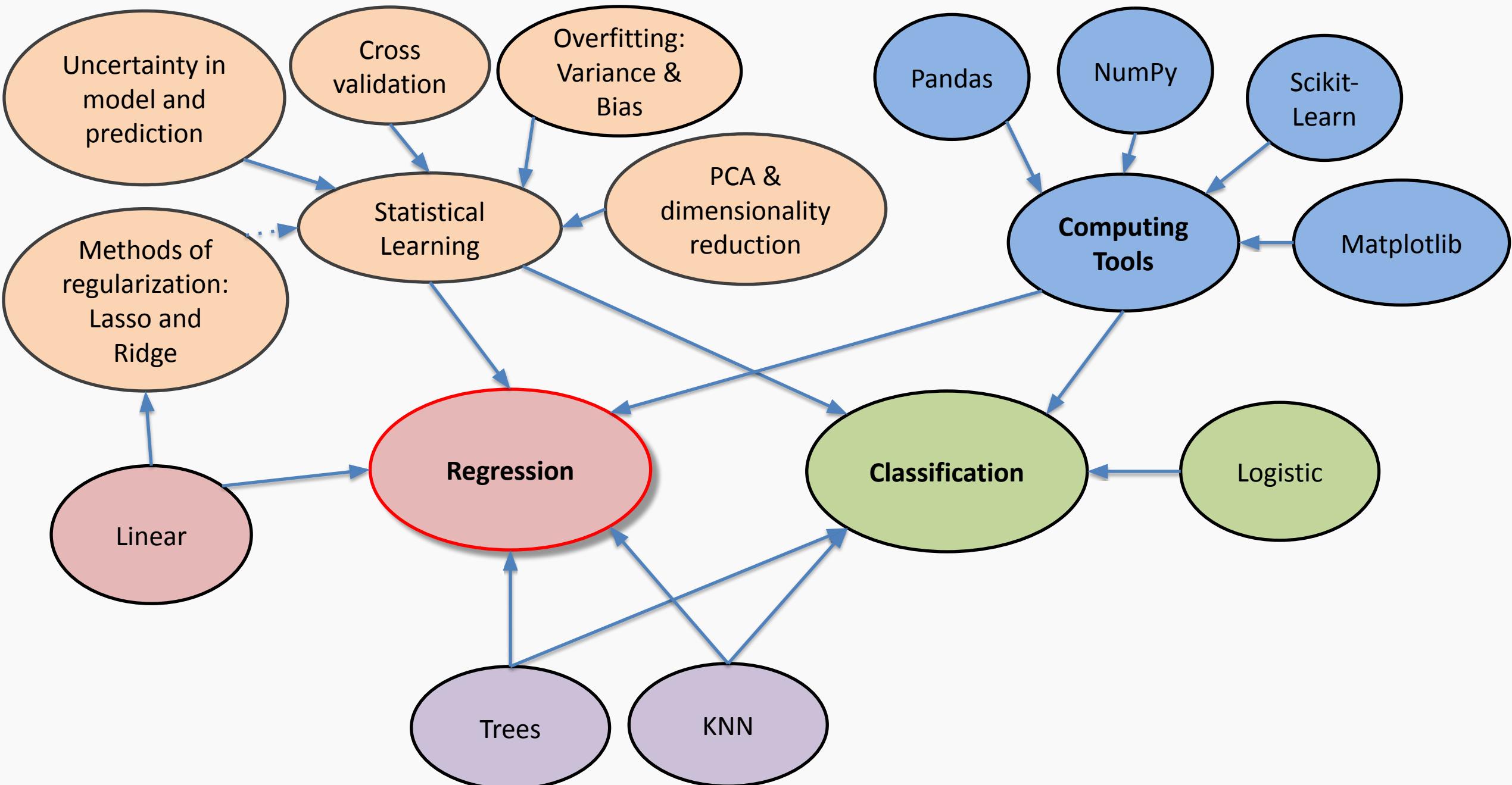
CS109A Introduction to Data Science
Pavlos Protopapas and Natesh Pillai



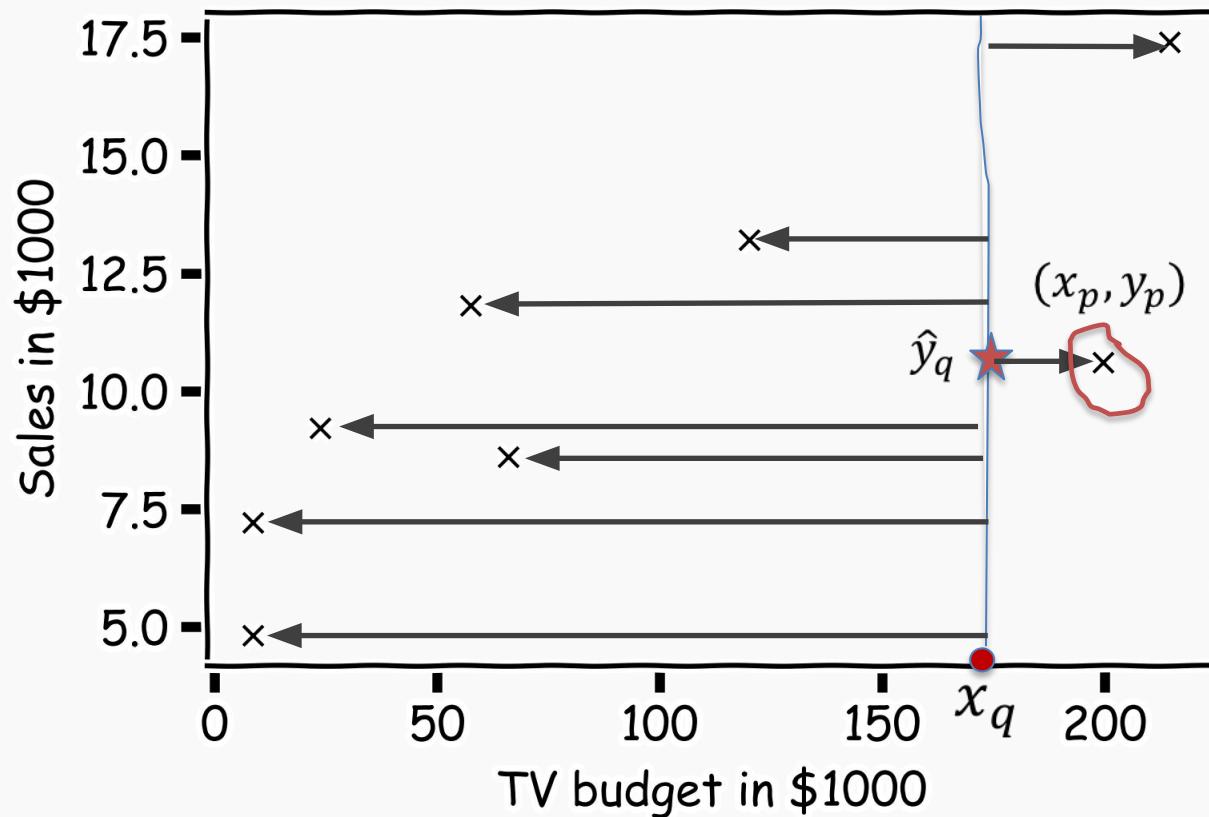
Lectures

-  **Lecture 1**
Due: Mon September 13th, 9:46 am GMT-04:00
-  **Lecture 2**
Due: Mon September 13th, 9:46 am GMT-04:00
-  **Lecture 3: Introduction to Regression - kNN and Linear Regression**
Due: Fri September 17th, 9:46 am GMT-04:00
-  **Lecture 4: Multi-linear and Polynomial Regression**
Due: Mon September 20th, 9:30 am GMT-04:00
-  **Lecture 5: Model Selection and Cross Validation**
Due: Fri September 24th, 9:46 am GMT-04:00
-  **Lecture 6: Regularization - Ridge and Lasso Regression**
Due: Mon September 27th, 9:46 am GMT-04:00
-  **Lecture 7: Probability**
Due: Mon October 4th, 9:46 am GMT-04:00
-  **Lecture 8: Inference in Regression and Hypothesis Testing**
Due: Mon October 4th, 9:46 am GMT-04:00
-  **Lecture 9: Missing Data & Imputation**
Due: Fri October 8th, 9:46 am GMT-04:00
-  **Lecture 10: Principal Component Analysis**
Due: Mon October 11th, 9:45 am GMT-04:00
-  **Lecture 11: Case Study**
-  **Lecture 12: Visualization**
Due: Mon October 25th, 9:46 am GMT-04:00
-  **Lecture 13: Ethics**
Due: Mon October 25th, 9:46 am GMT-04:00
-  **Lecture 14 Logistic Regression 1**
Due: Fri October 29th, 9:46 am GMT-04:00
-  **Lecture 15 Logistic Regression 2**
Due: Mon November 1st, 9:46 am GMT-04:00
-  **Lecture 16: Decision Trees**
Due: Fri November 5th, 9:46 am GMT-04:00
-  **Lecture 17: Bagging**
Due: Mon November 8th, 10:00 am GMT-05:00
-  **Lecture 18: Random Forest**
Due: Fri November 19th, 10:00 am GMT-05:00
-  **Lecture 19: Random Forrest II**
Due: Mon November 13th, 10:00 am GMT-05:00
-  **Lecture 20: Boosting, Gradient Boosting**
Due: Fri November 19th, 10:00 am GMT-05:00
-  **Lecture 21: Ada Boost**
Due: Mon November 22nd, 10:00 am GMT-05:00
-  **Lecture 22: Working Example**
-  **Lecture 23: Natural Language Processing**
Due: Fri December 3rd, 10:00 am GMT-05:00





Simple Prediction Model (KNN)



What is \hat{y}_q at some x_q ?

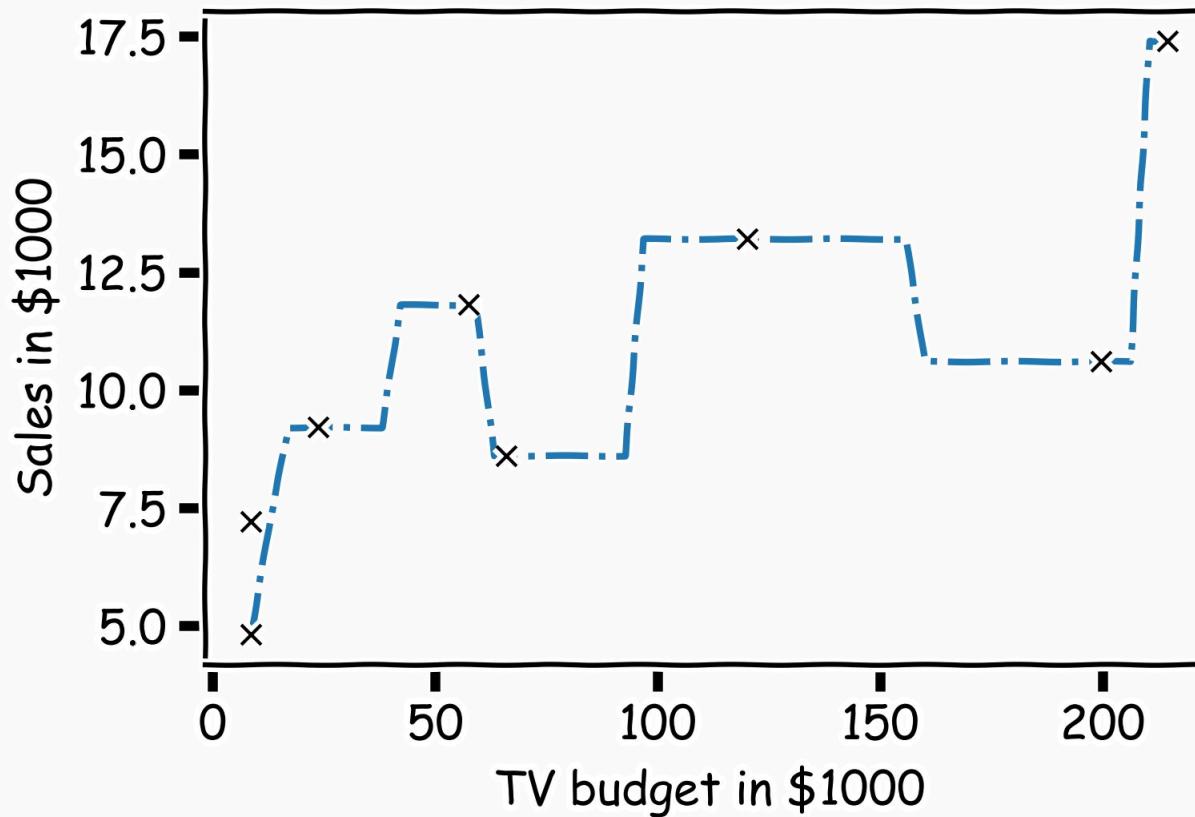
Find distances to all other points $D(x_q, x_i)$

Find the nearest neighbor, (x_p, y_p)

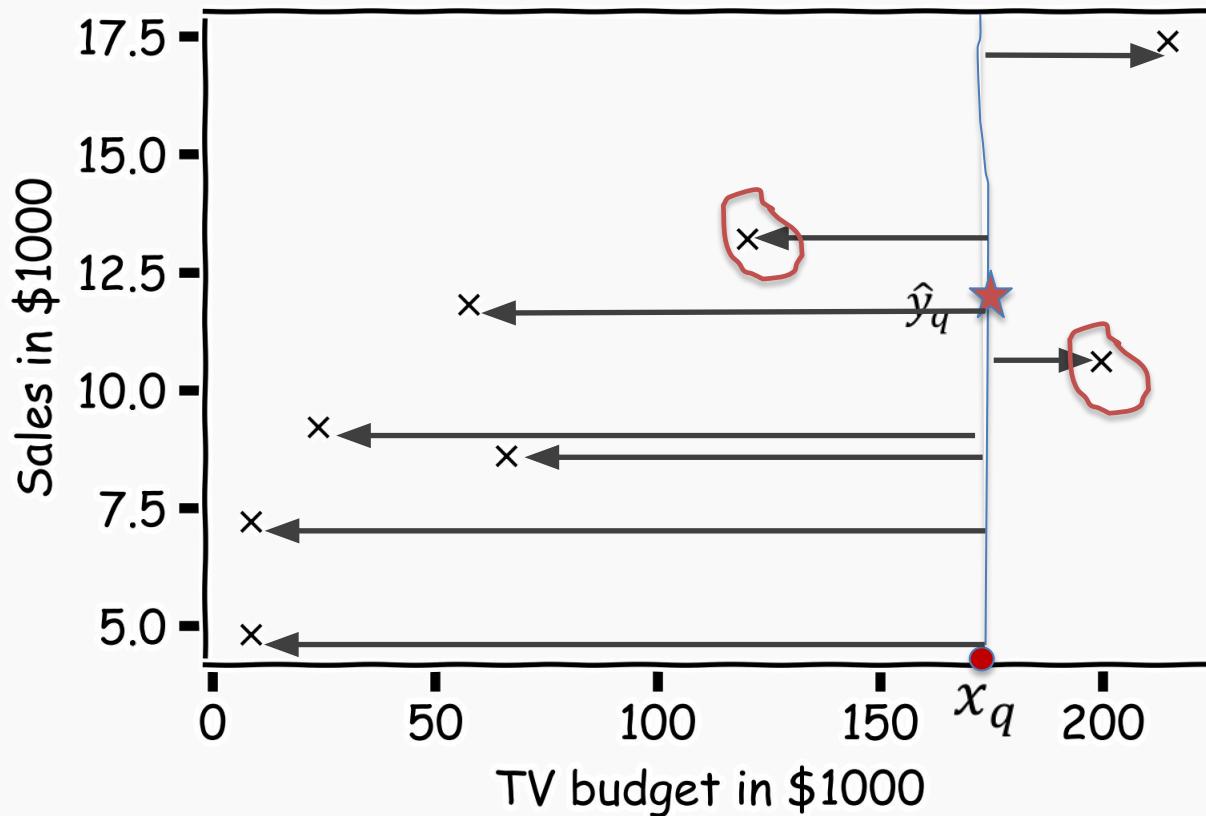
Predict $\hat{y}_q = y_p$

Simple Prediction Model

Do the same for “all” x 's



Extend the Prediction Model



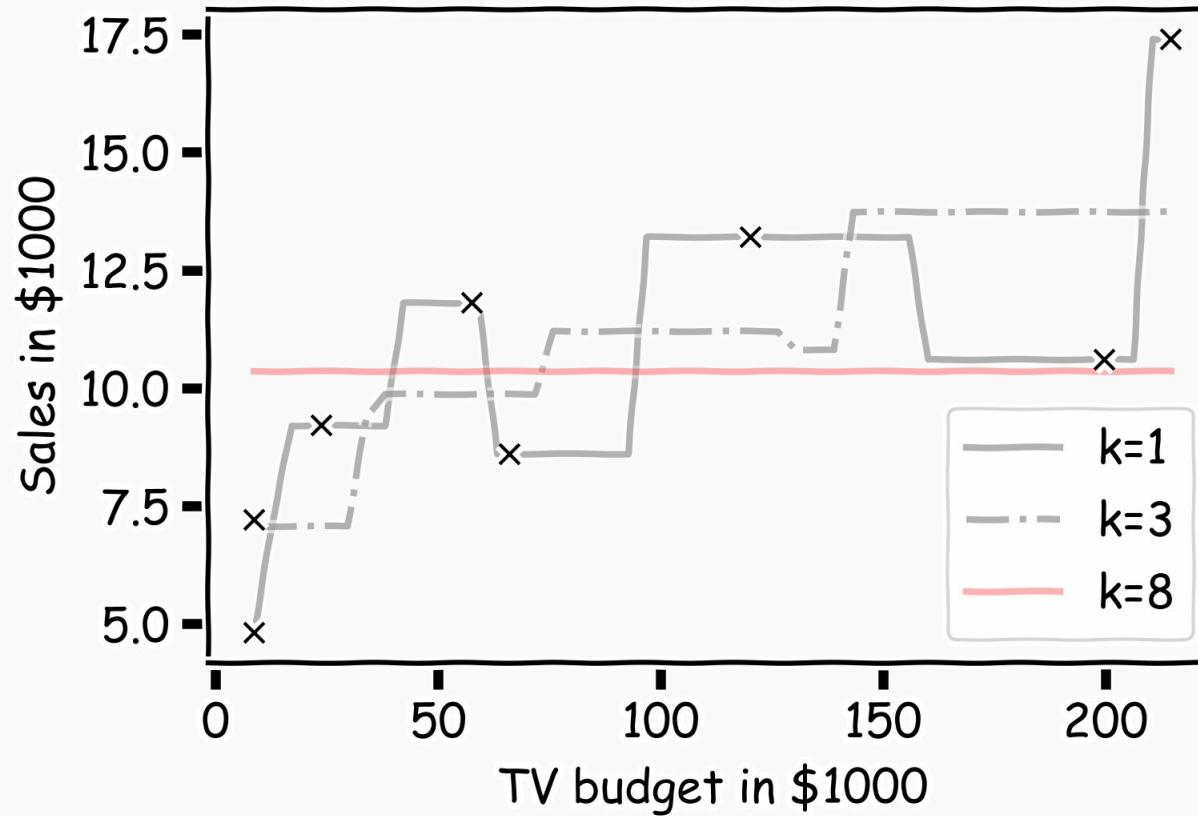
What is \hat{y}_q at some x_q ?

Find distances to all other points $D(x_q, x_i)$

Find the k-nearest neighbors, x_{q_1}, \dots, x_{q_k}

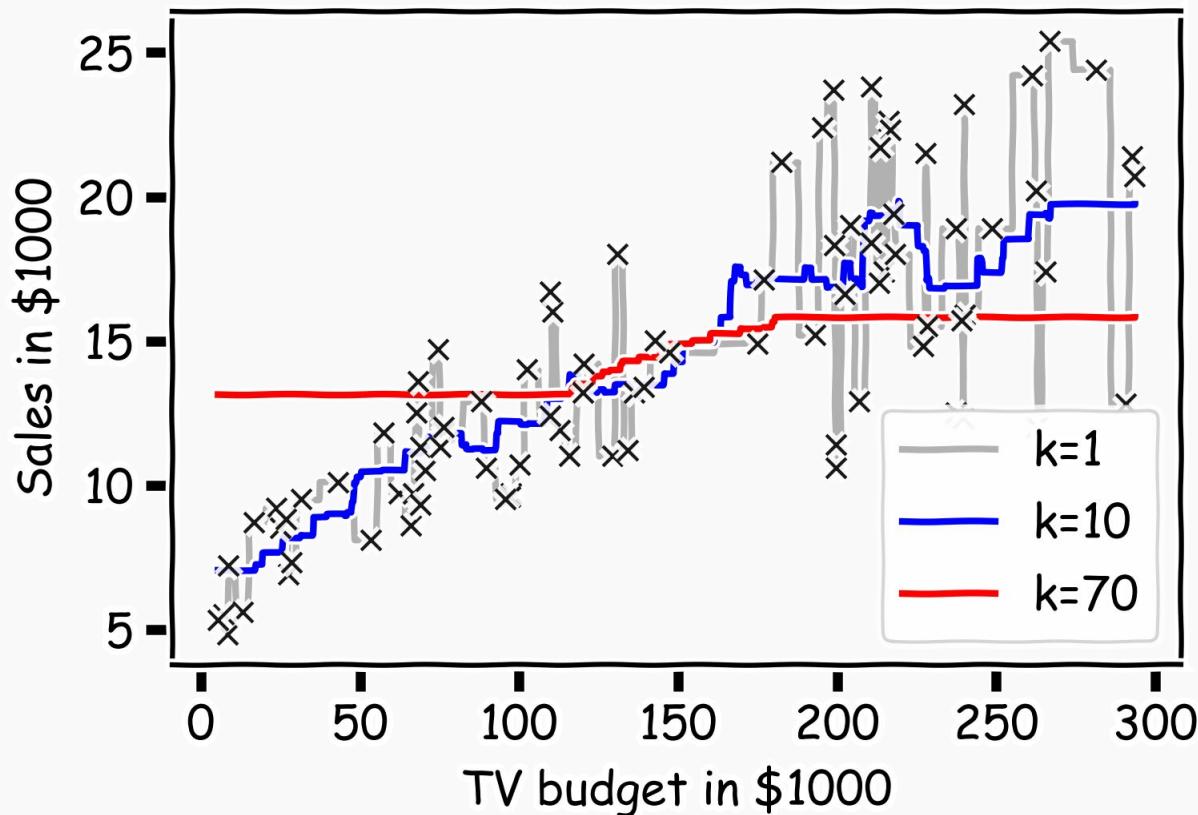
Predict $\hat{y}_q = \frac{1}{k} \sum_i^k y_{q_i}$

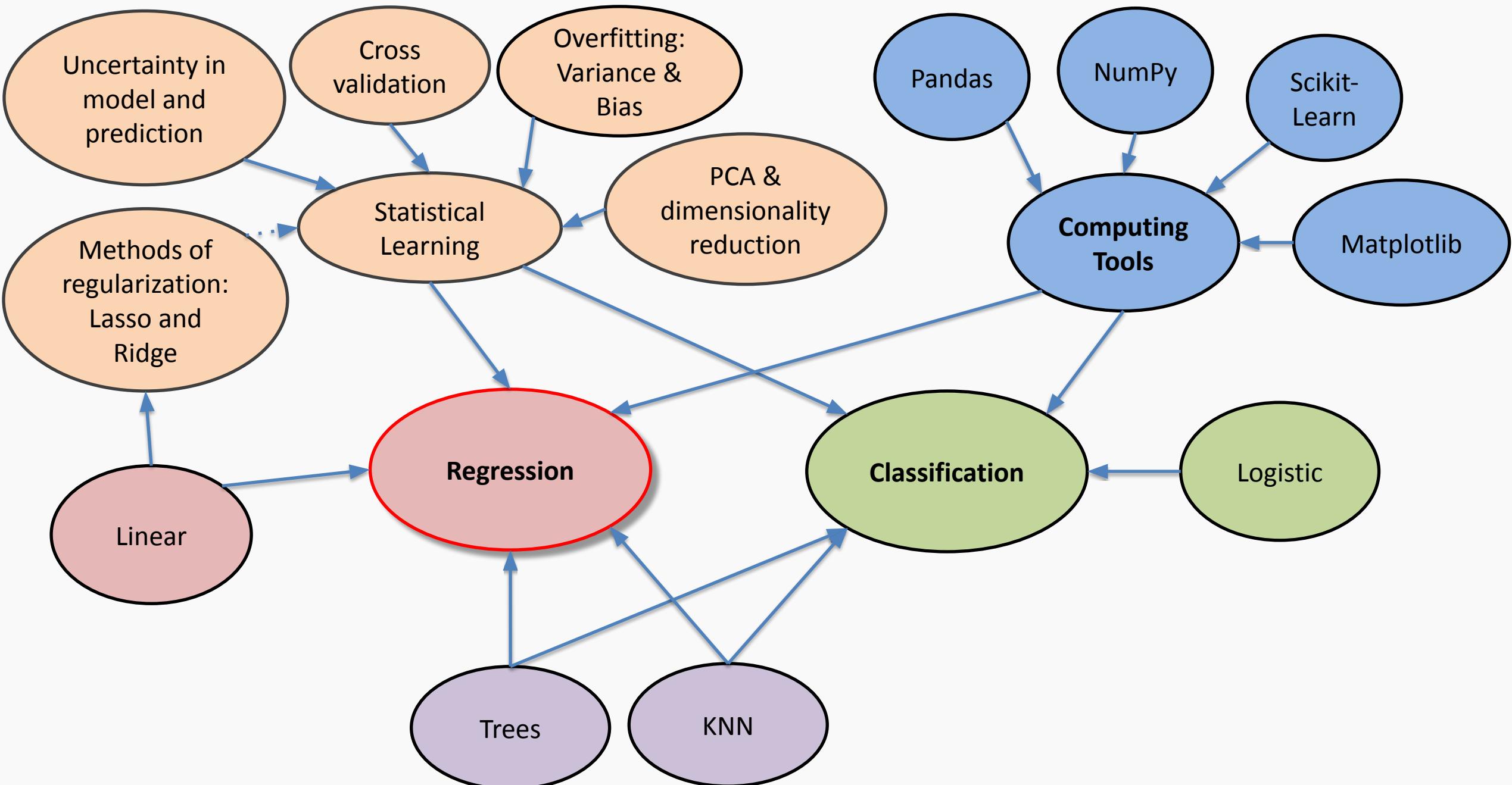
Simple Prediction Models

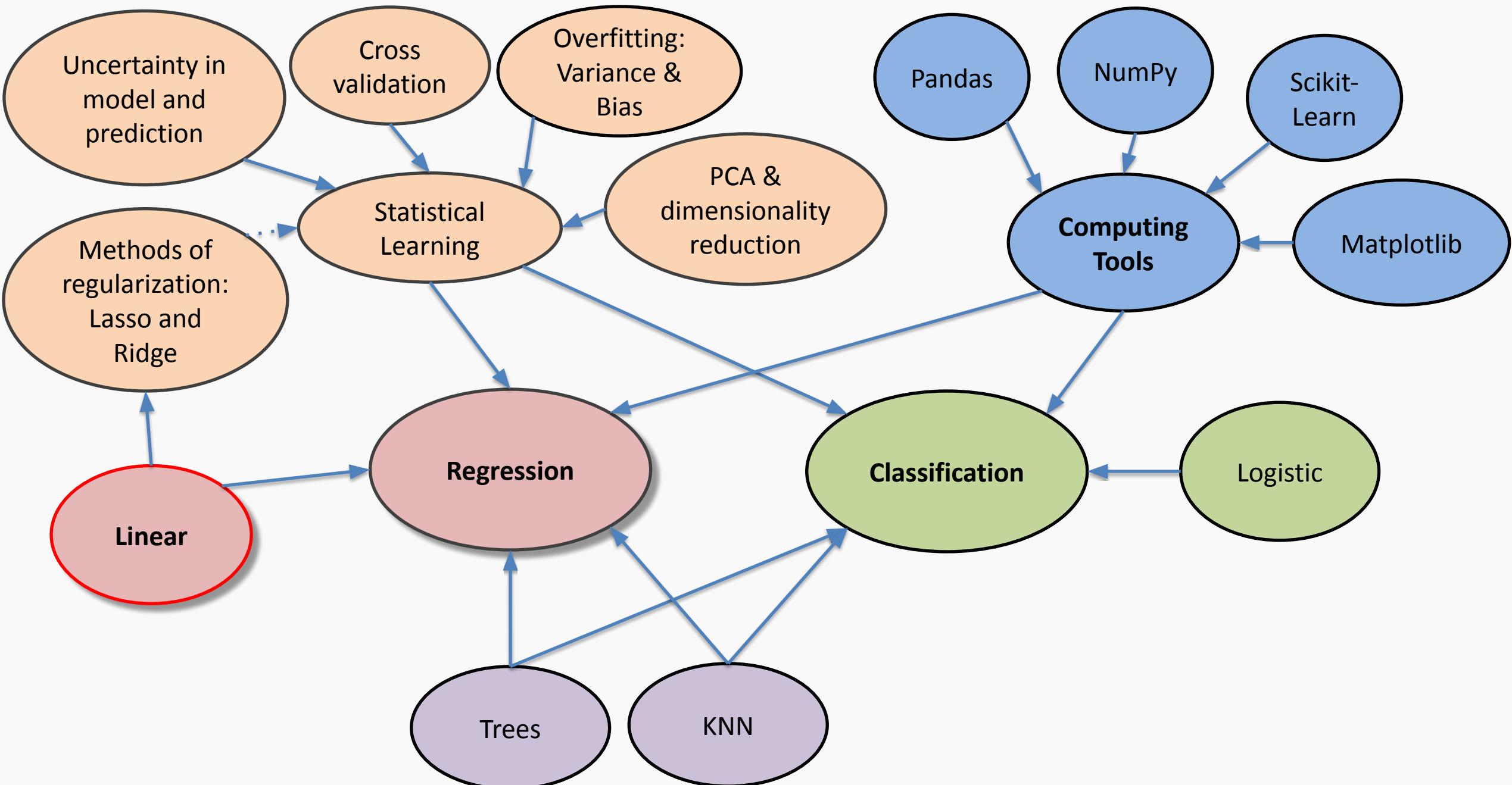


Simple Prediction Models

We can try different k-models on more data

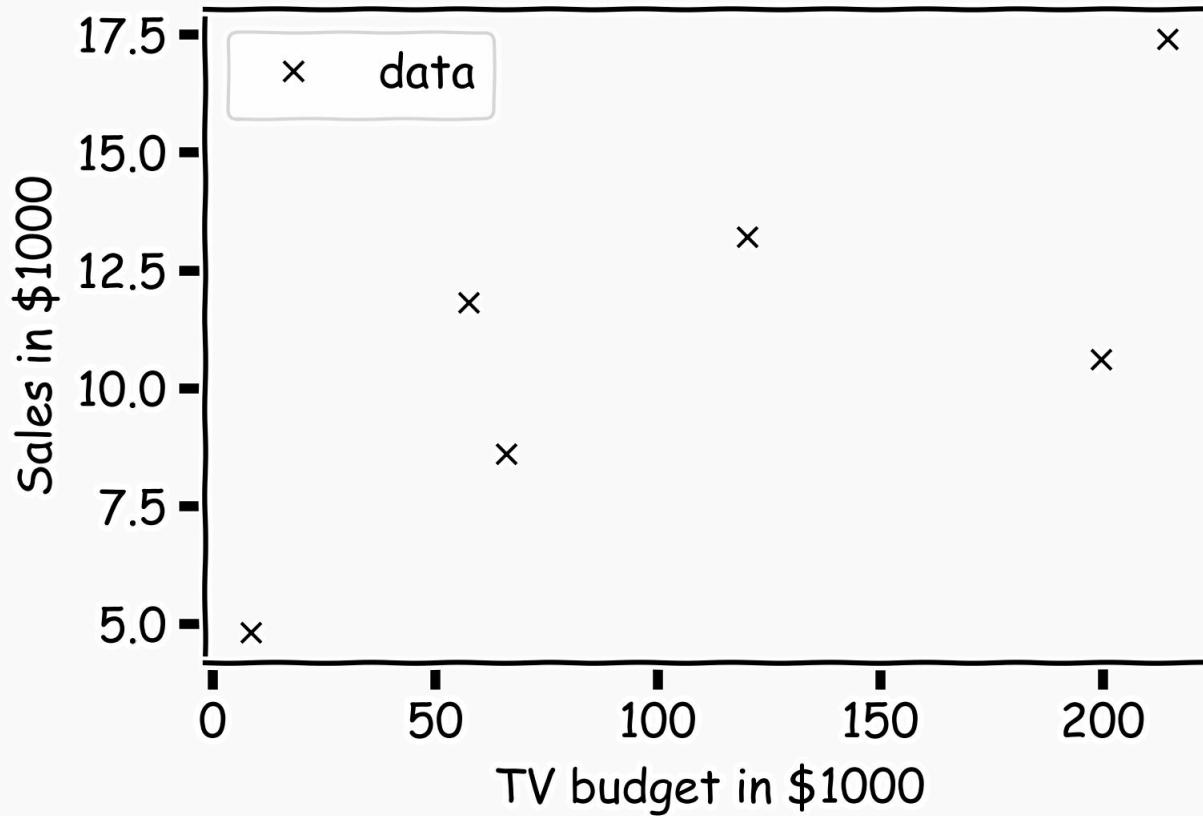






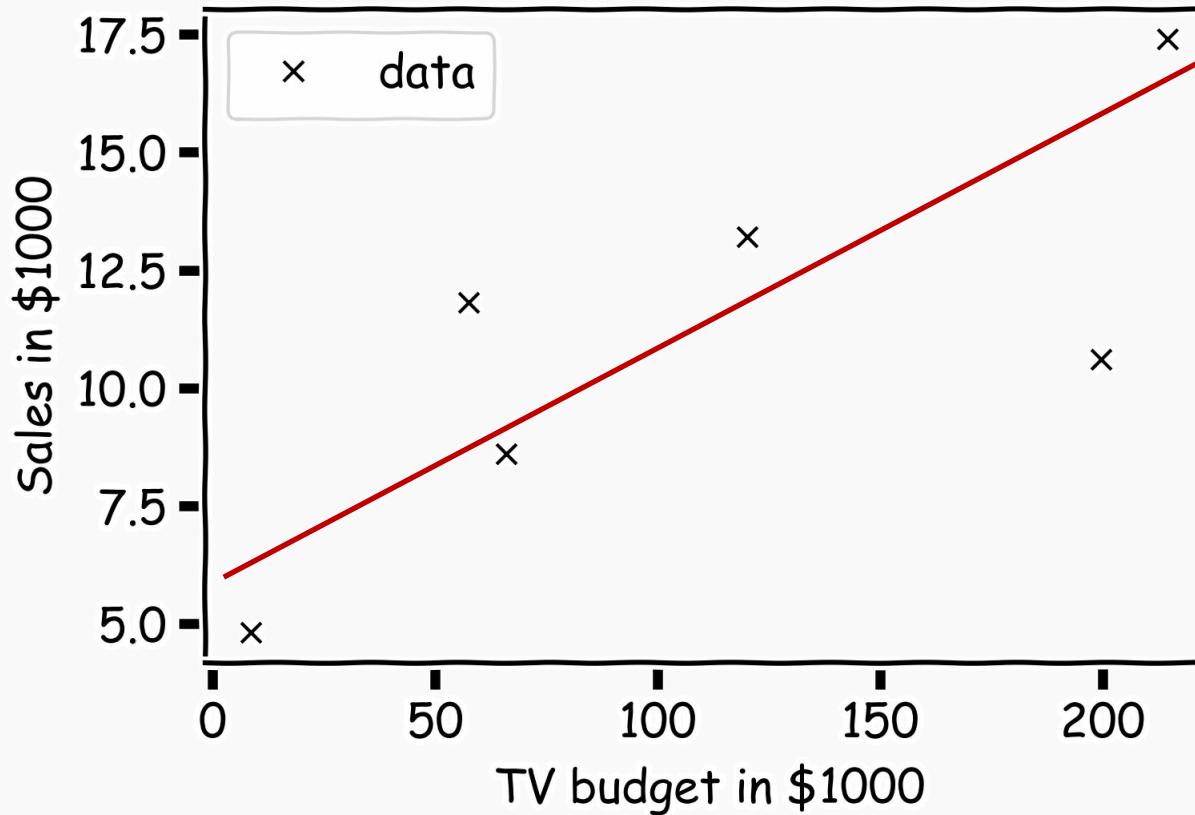
Estimate of the regression coefficients

For a given data set



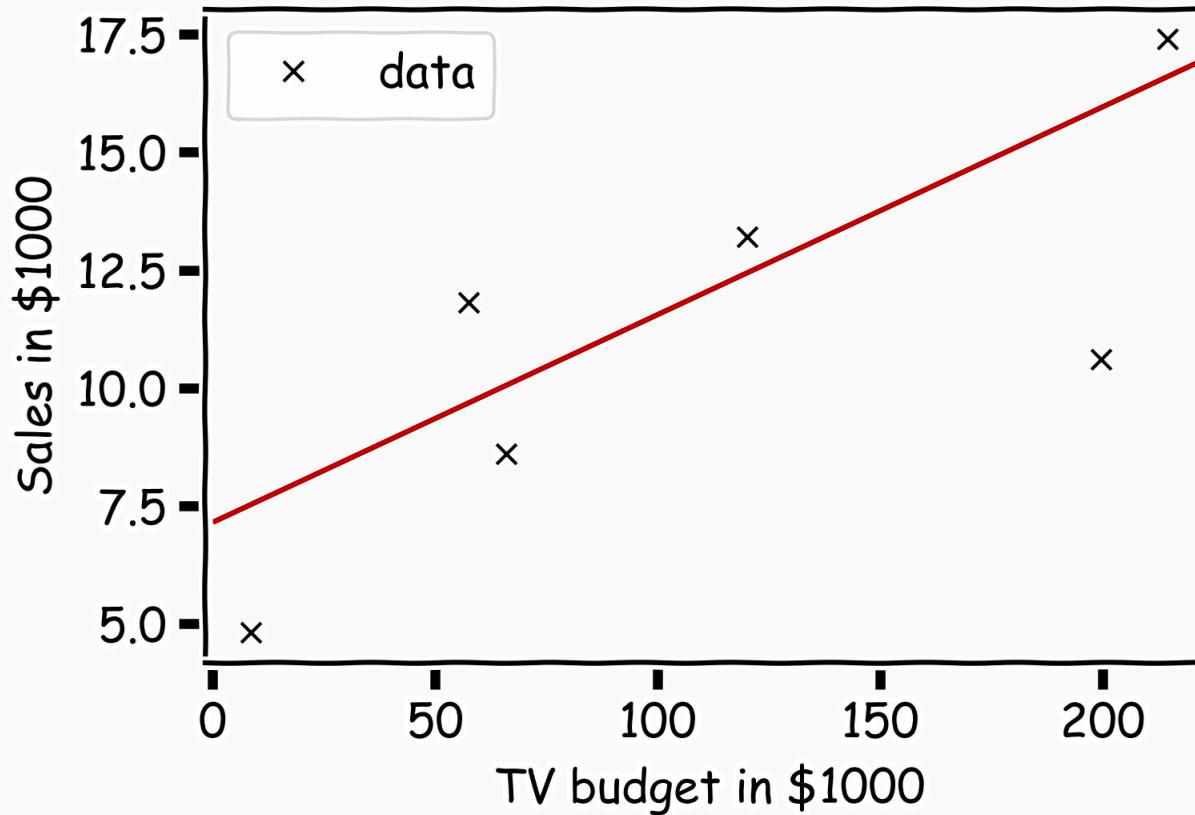
Estimate of the regression coefficients (cont)

Is this line good?



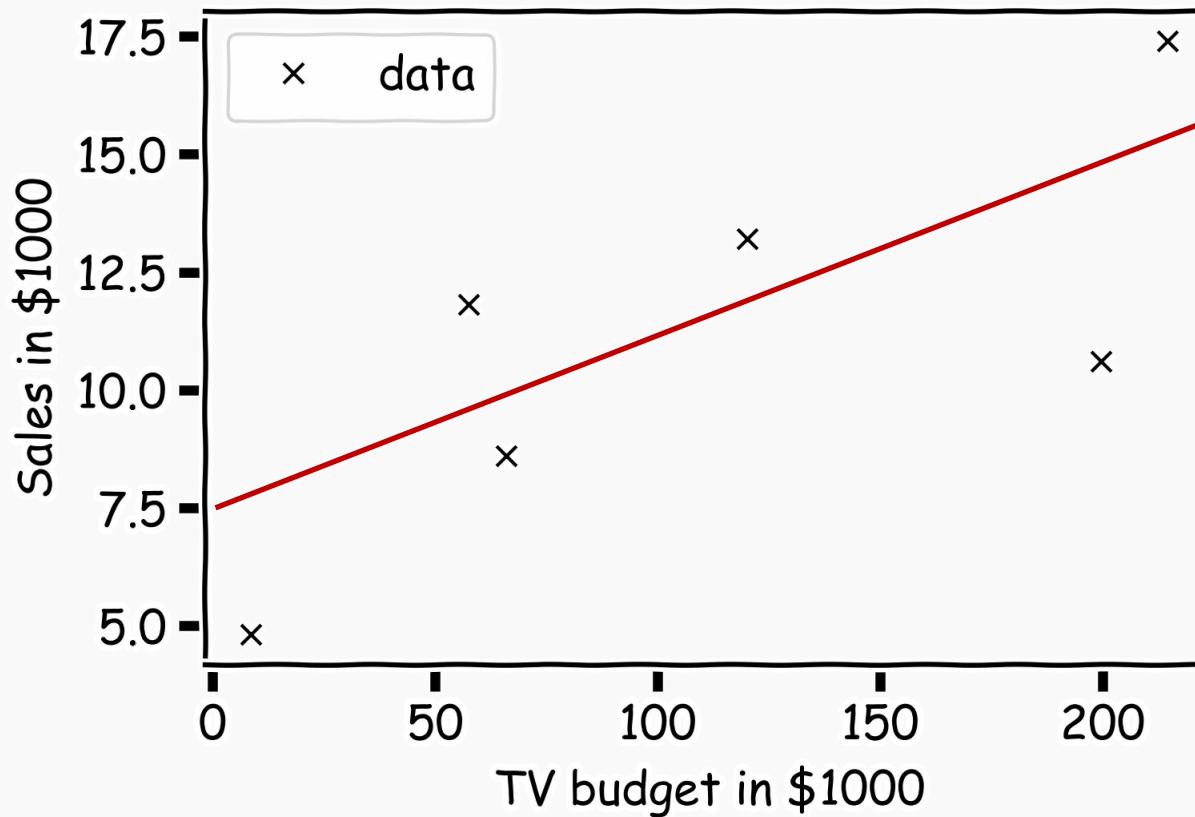
Estimate of the regression coefficients (cont)

Maybe this one?



Estimate of the regression coefficients (cont)

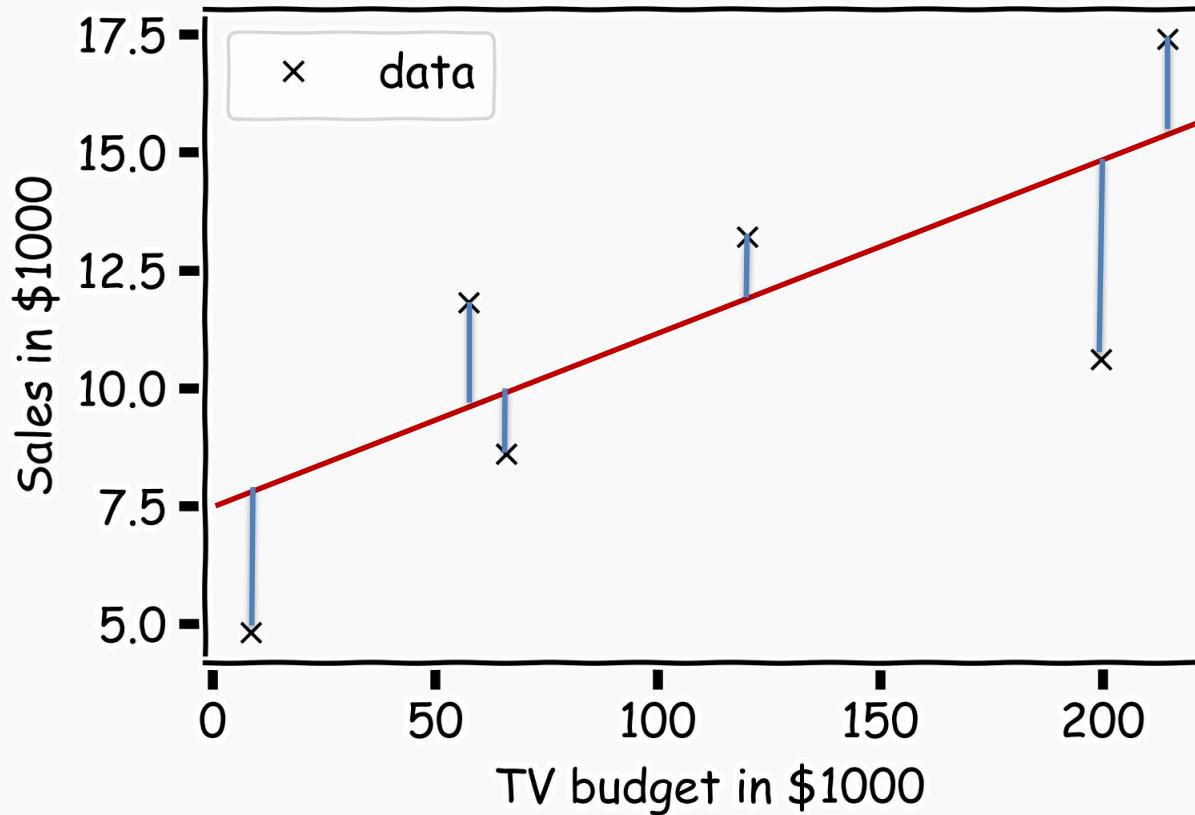
Or this one?



Estimate of the regression coefficients (cont)

Question: Which line is the best?

First calculate the residuals



Estimate of the regression coefficients (cont)

Again we use MSE as our **loss function**,

$$L(\beta_0, \beta_1) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \frac{1}{n} \sum_{i=1}^n [y_i - (\beta_1 X + \beta_0)]^2.$$

We choose $\hat{\beta}_1$ and $\hat{\beta}_0$ in order to minimize the predictive errors made by our model, i.e. minimize our loss function.

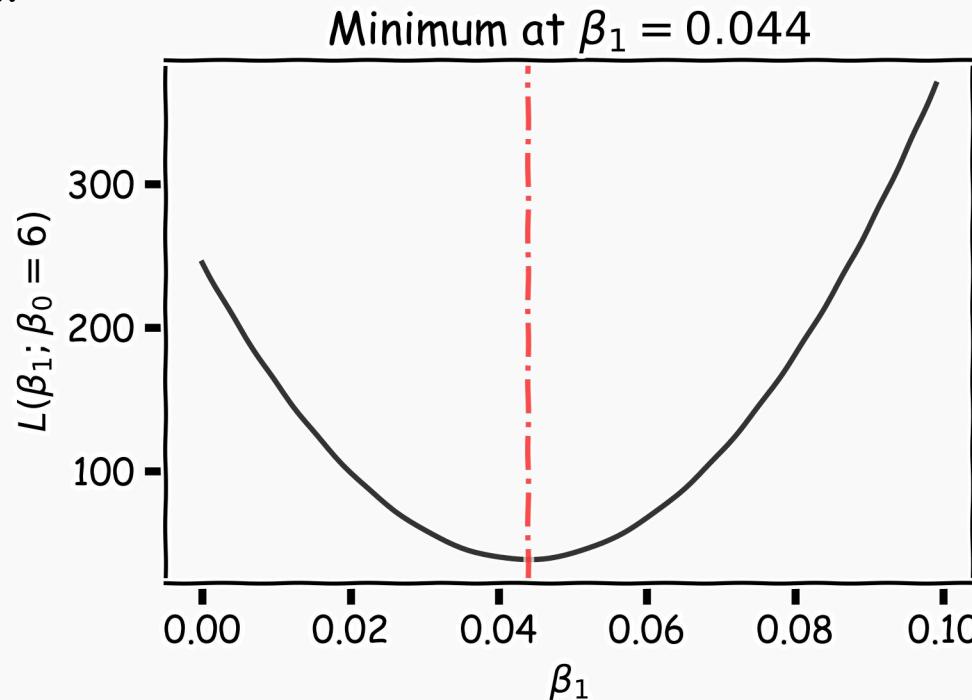
Then the optimal values for $\hat{\beta}_0$ and $\hat{\beta}_1$ should be:

$$\hat{\beta}_0, \hat{\beta}_1 = \operatorname{argmin}_{\beta_0, \beta_1} L(\beta_0, \beta_1).$$

Estimate of the regression coefficients: brute force

One way to estimate our coefficients would be to calculate the loss function for every possible β_0 and β_1 . Then select the betas where the loss function is at the minimum.

E.g. the loss function for different β_1 values when β_0 is fixed to be 6:



Estimate of the regression coefficients: exact method

Take the partial derivatives of L with respect to β_0 and β_1 , set to zero, and find the solution to that equation. This procedure will give us explicit formulae for $\hat{\beta}_0$ and $\hat{\beta}_1$:

$$\hat{\beta}_1 = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$$

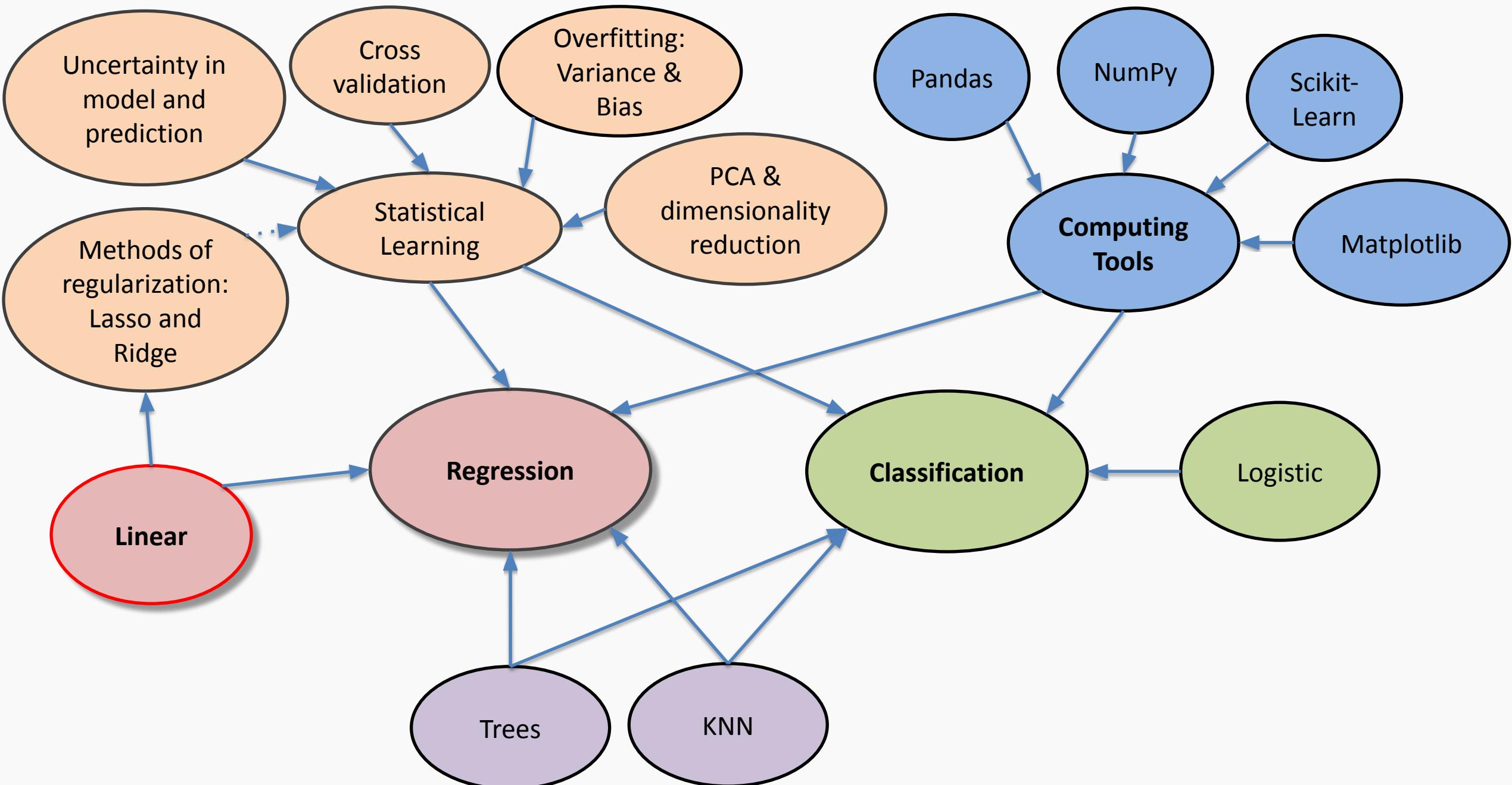
$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

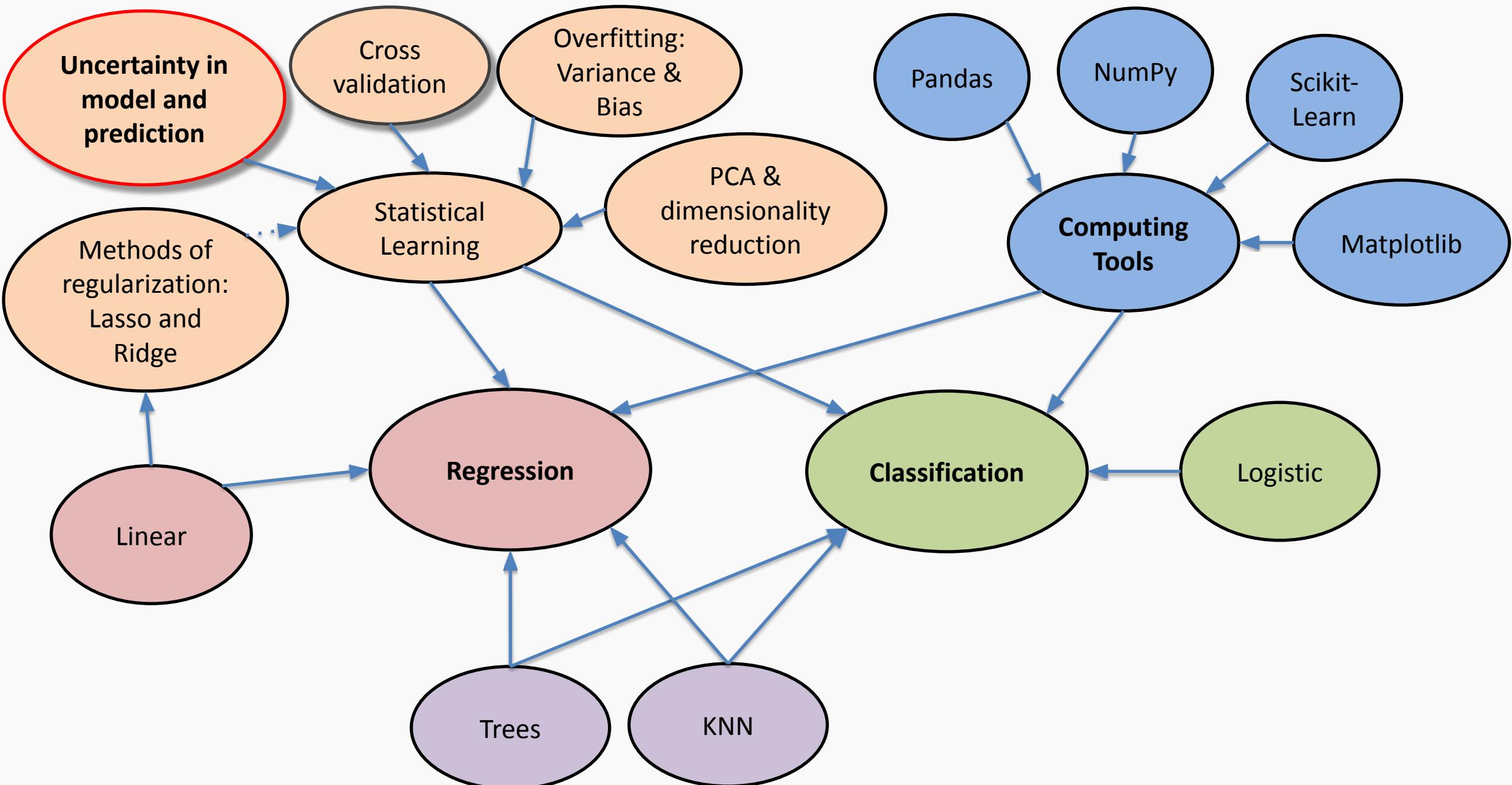
where \bar{y} and \bar{x} are sample means.

The line:

$$\hat{Y} = \hat{\beta}_1 X + \hat{\beta}_0$$

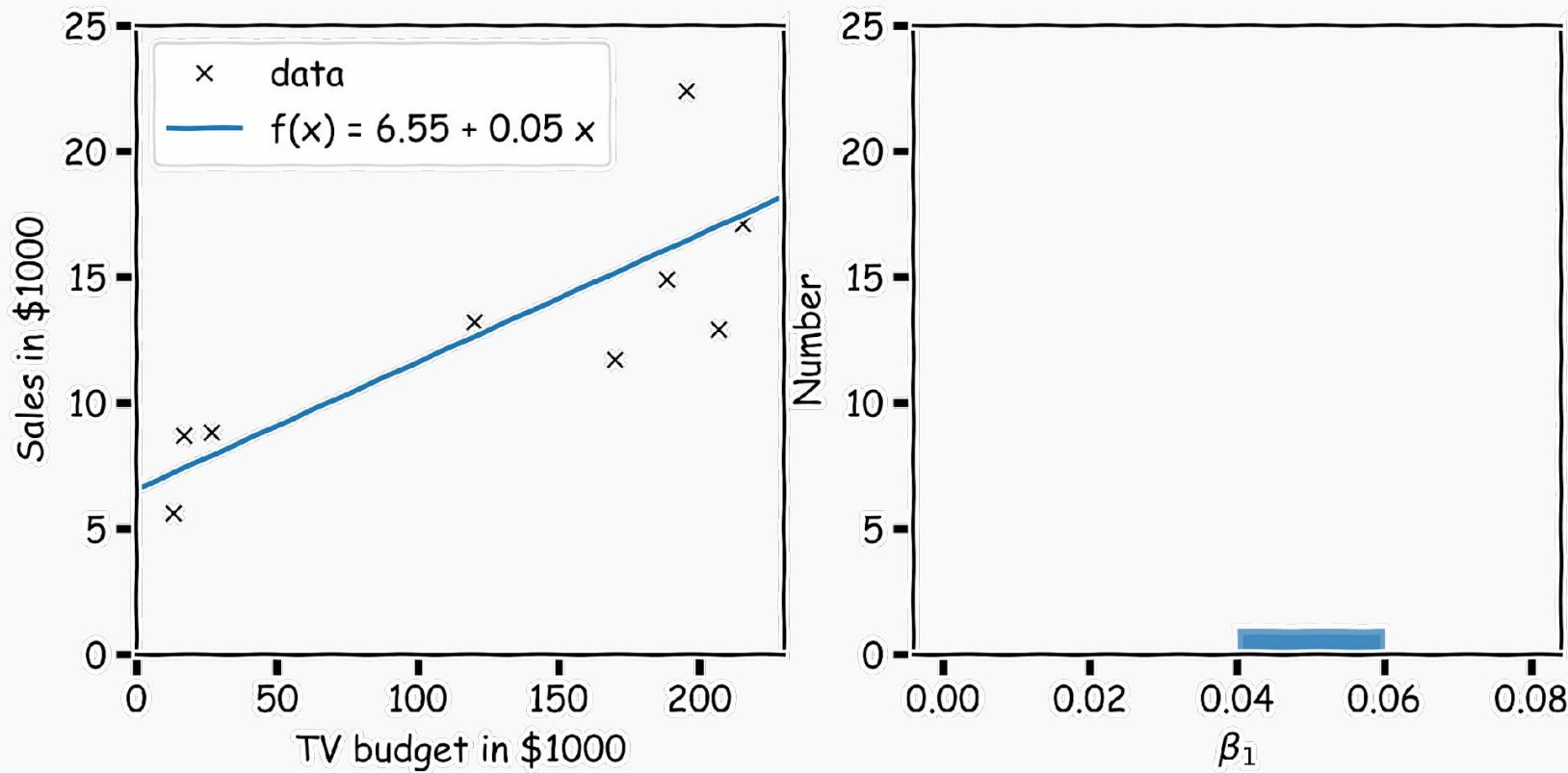
is called the **regression line**.





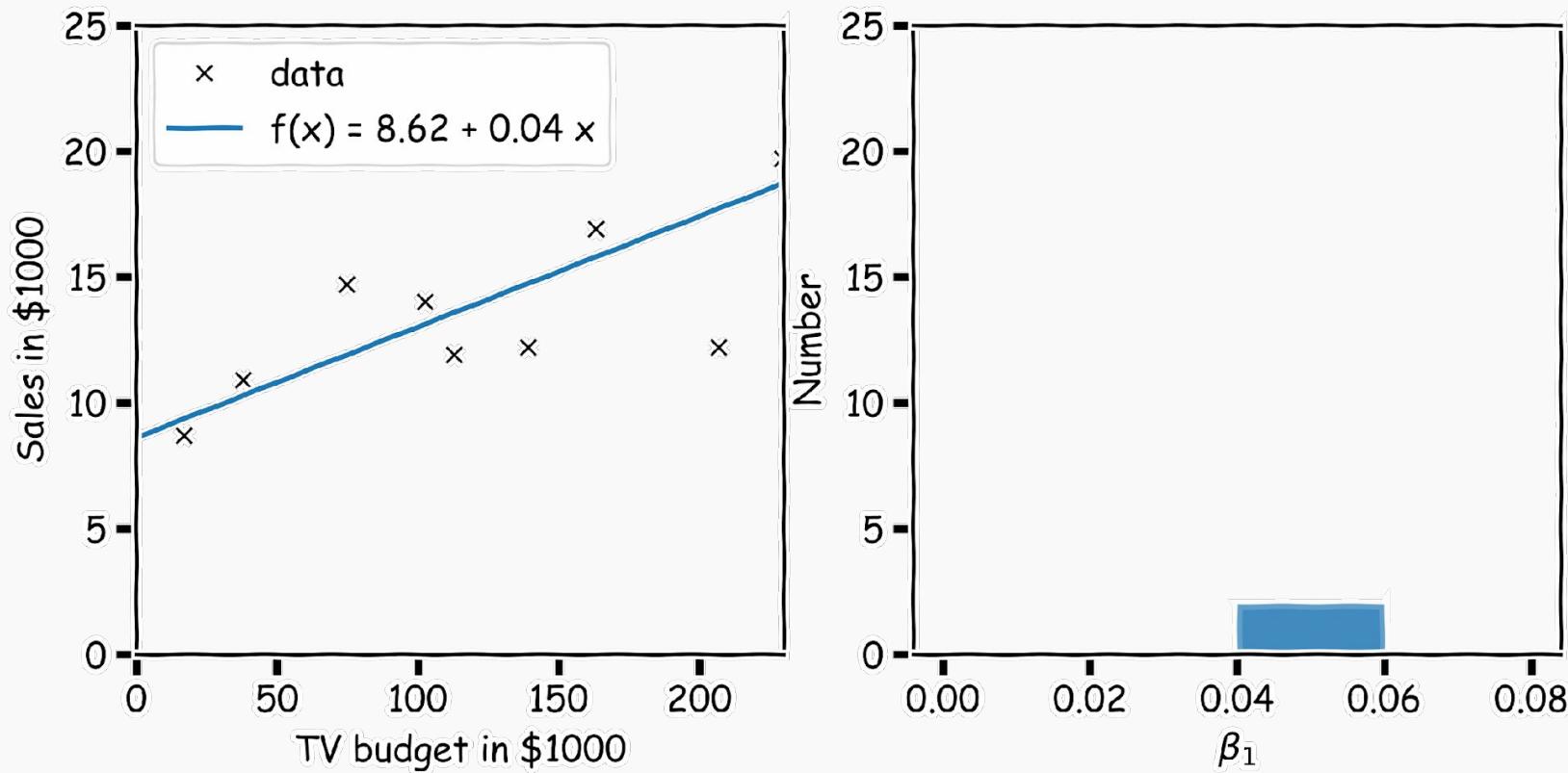
Confidence intervals for the predictors estimates

In our magical realisms, we can now sample multiple times



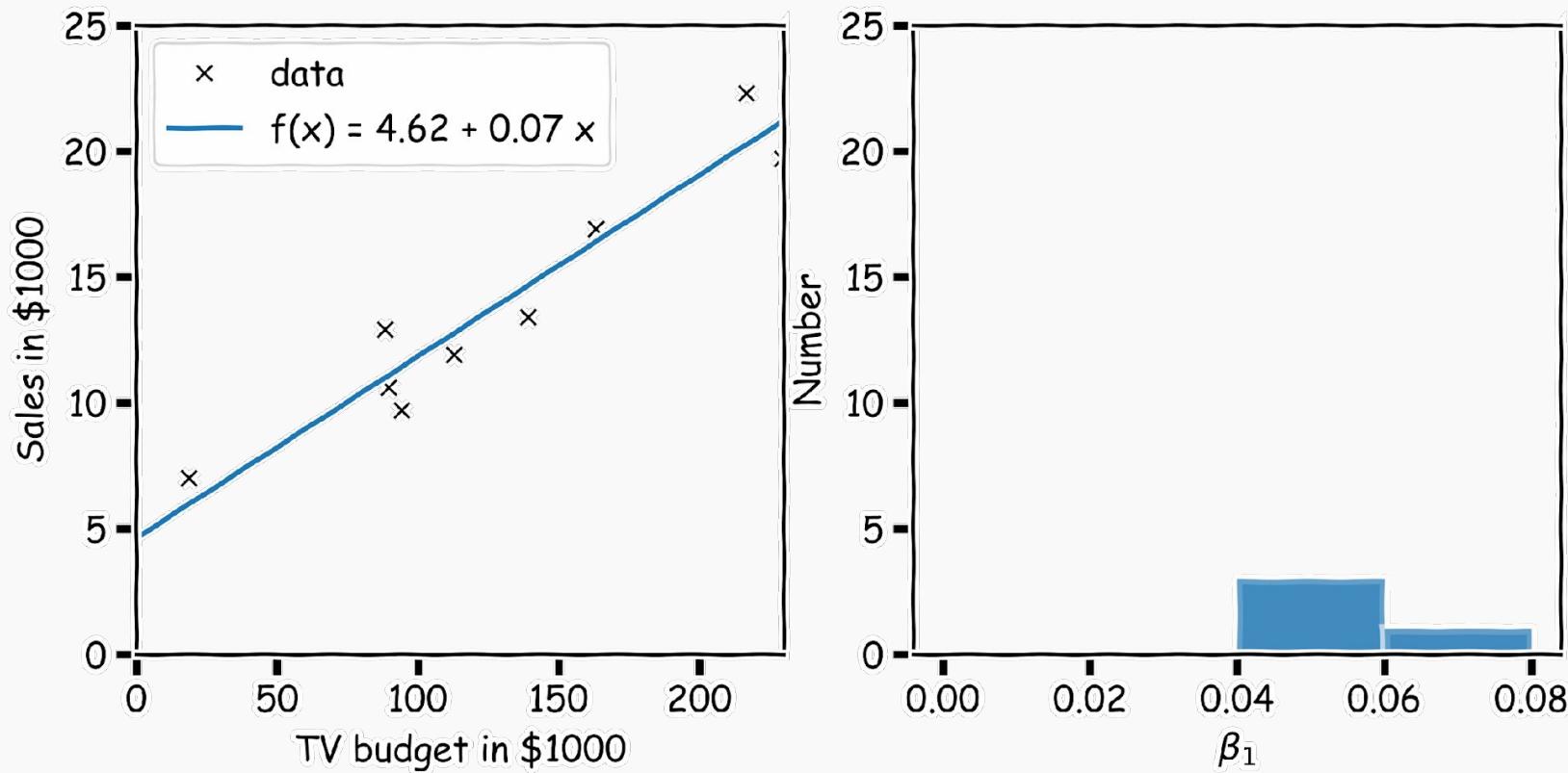
Confidence intervals for the predictors estimates (cont)

Another sample



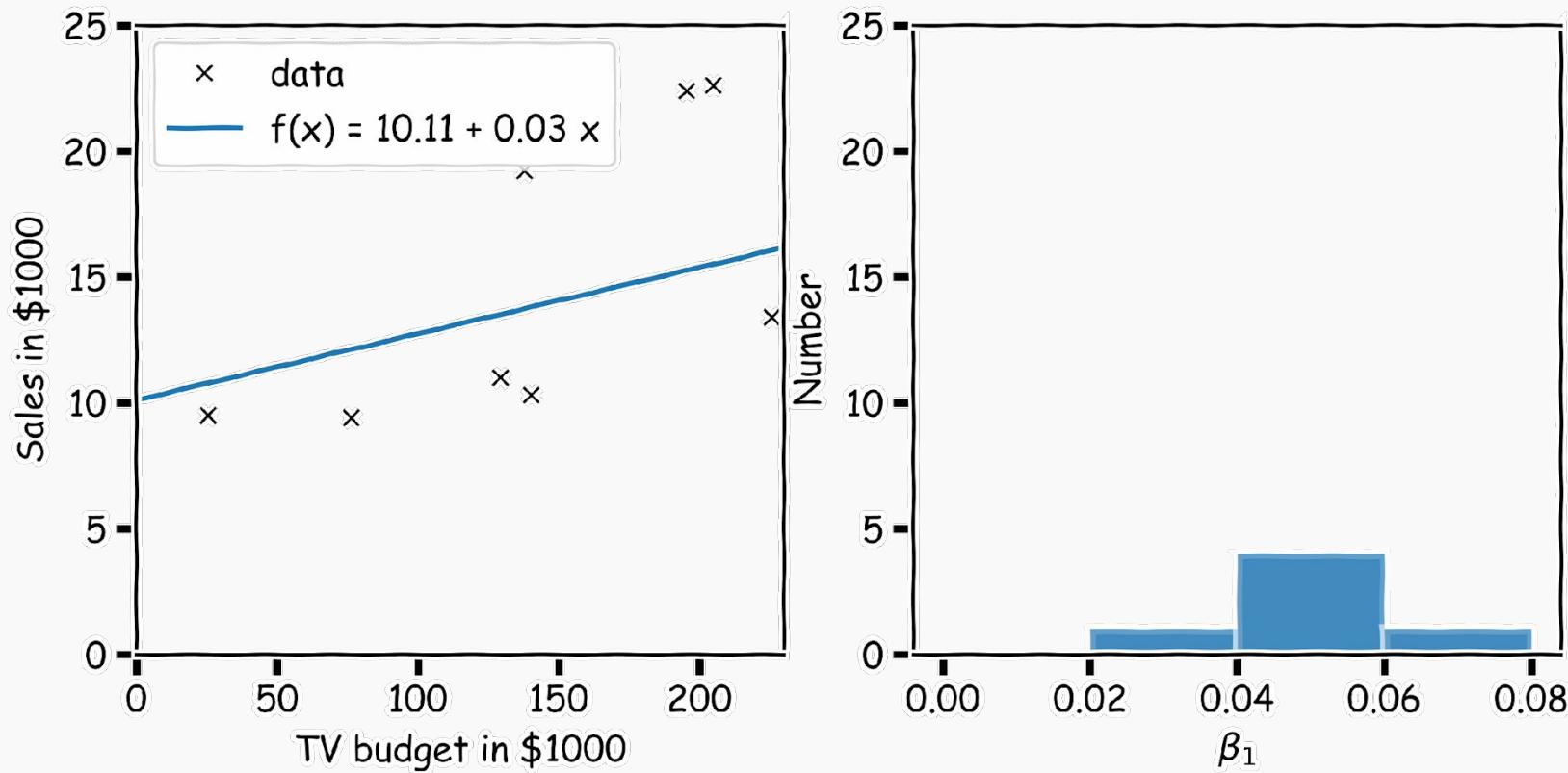
Confidence intervals for the predictors estimates (cont)

Another sample



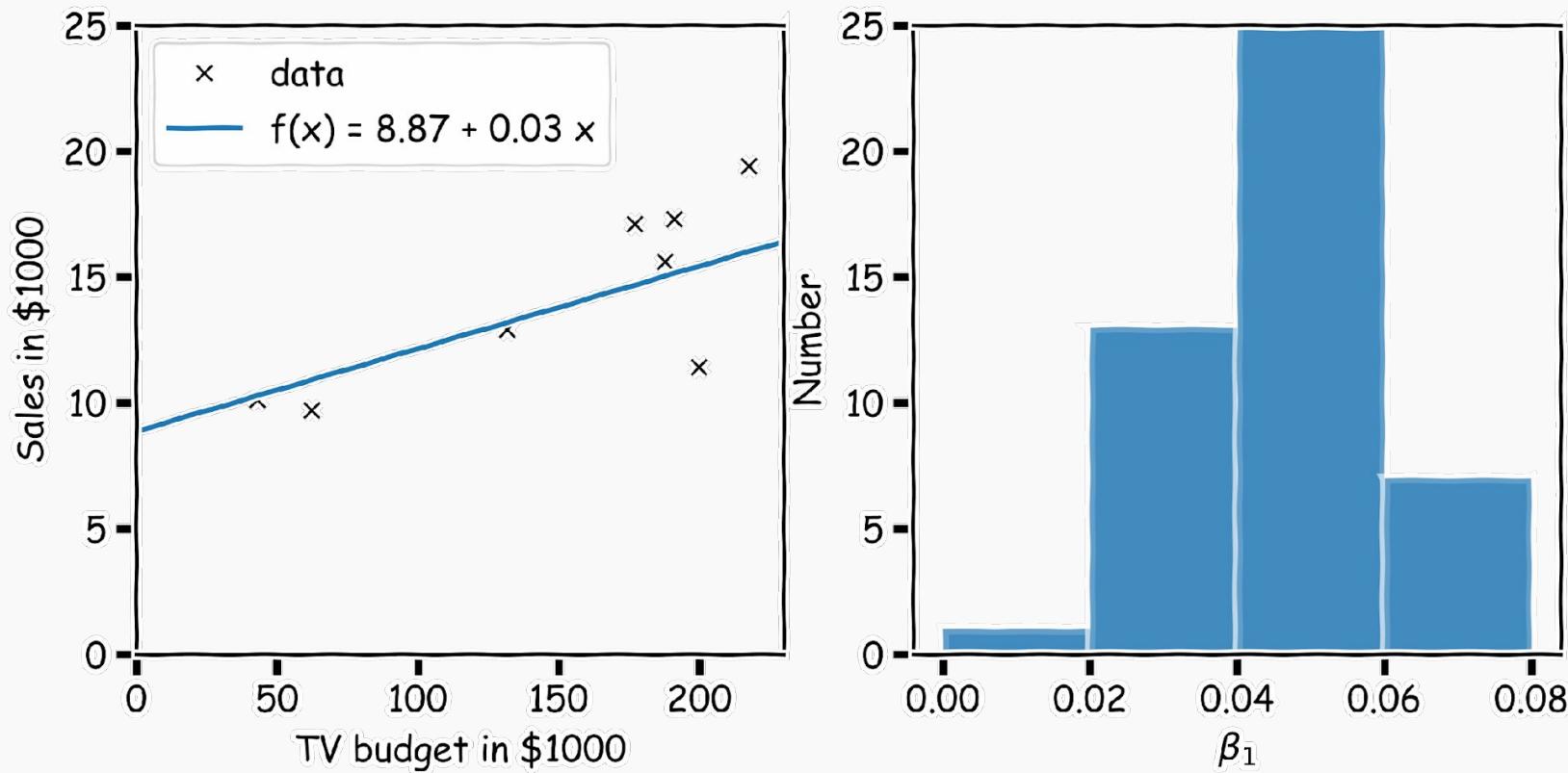
Confidence intervals for the predictors estimates (cont)

And another sample



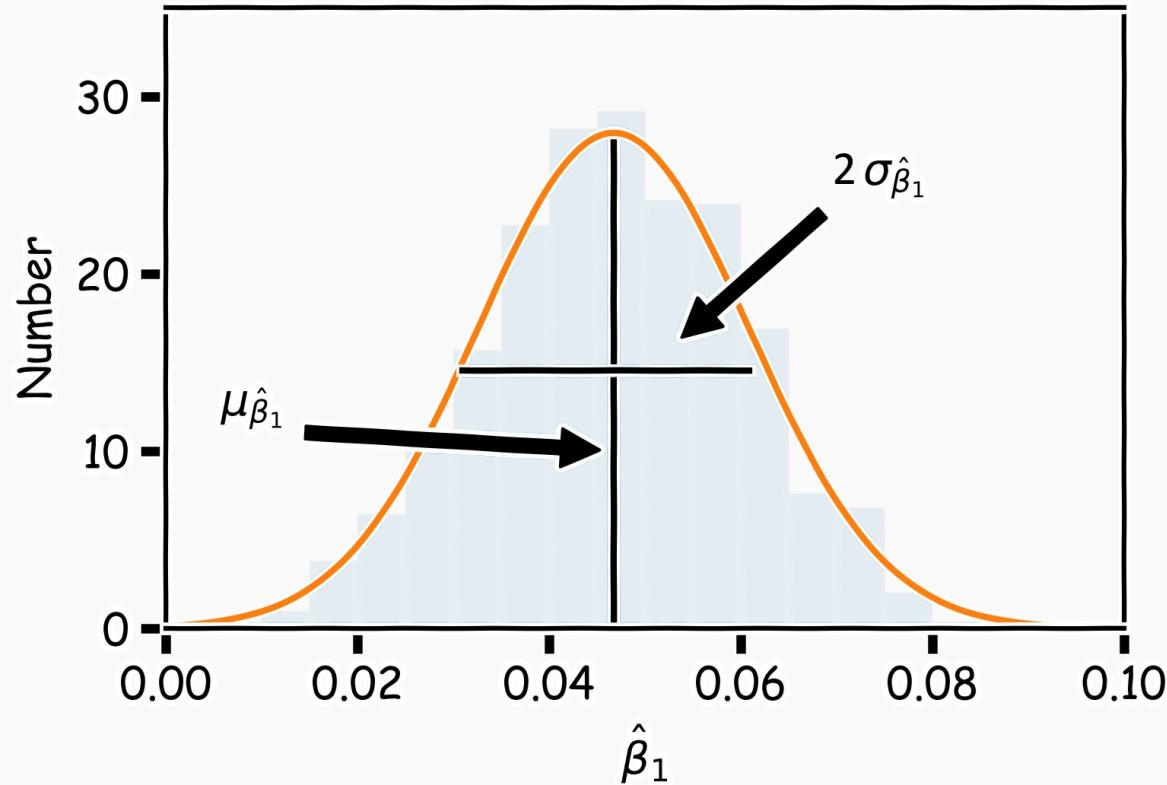
Confidence intervals for the predictors estimates (cont)

Repeat this for 100 times



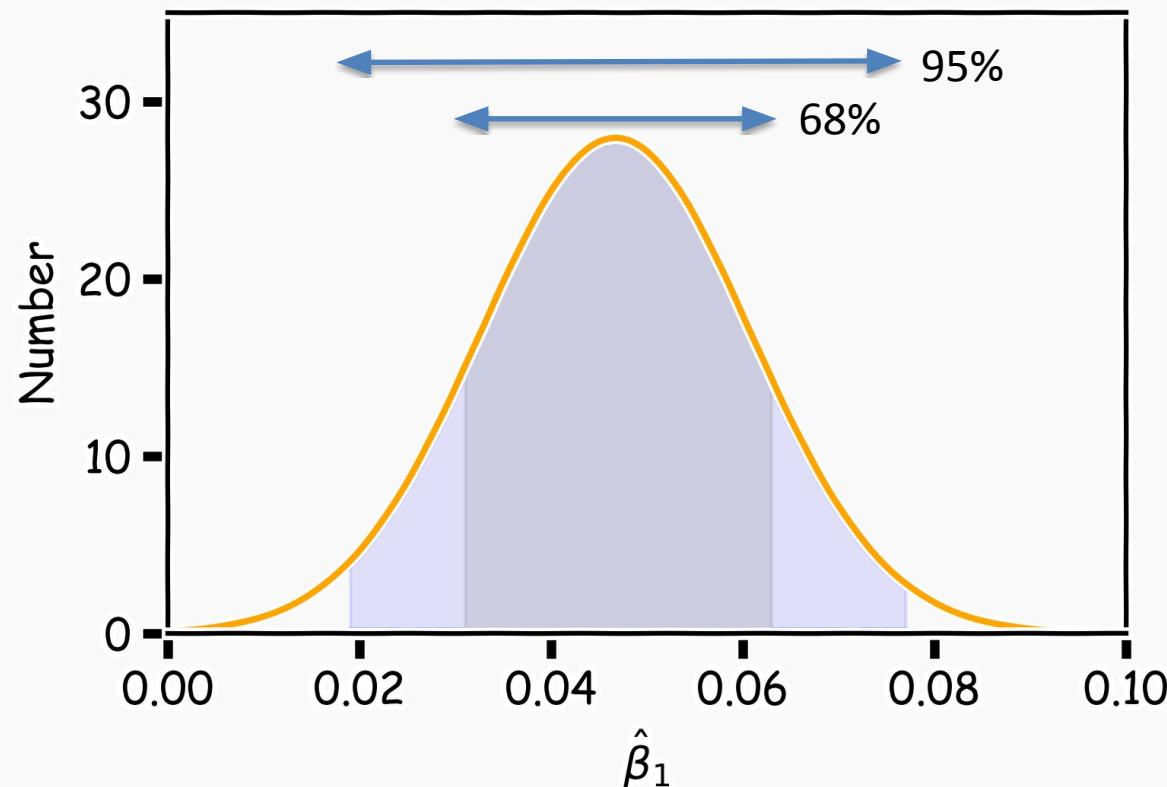
Confidence intervals for the predictors estimates (cont)

We can now estimate the mean and standard deviation of all the estimates $\hat{\beta}_1$.



Confidence intervals for the predictors estimates (cont)

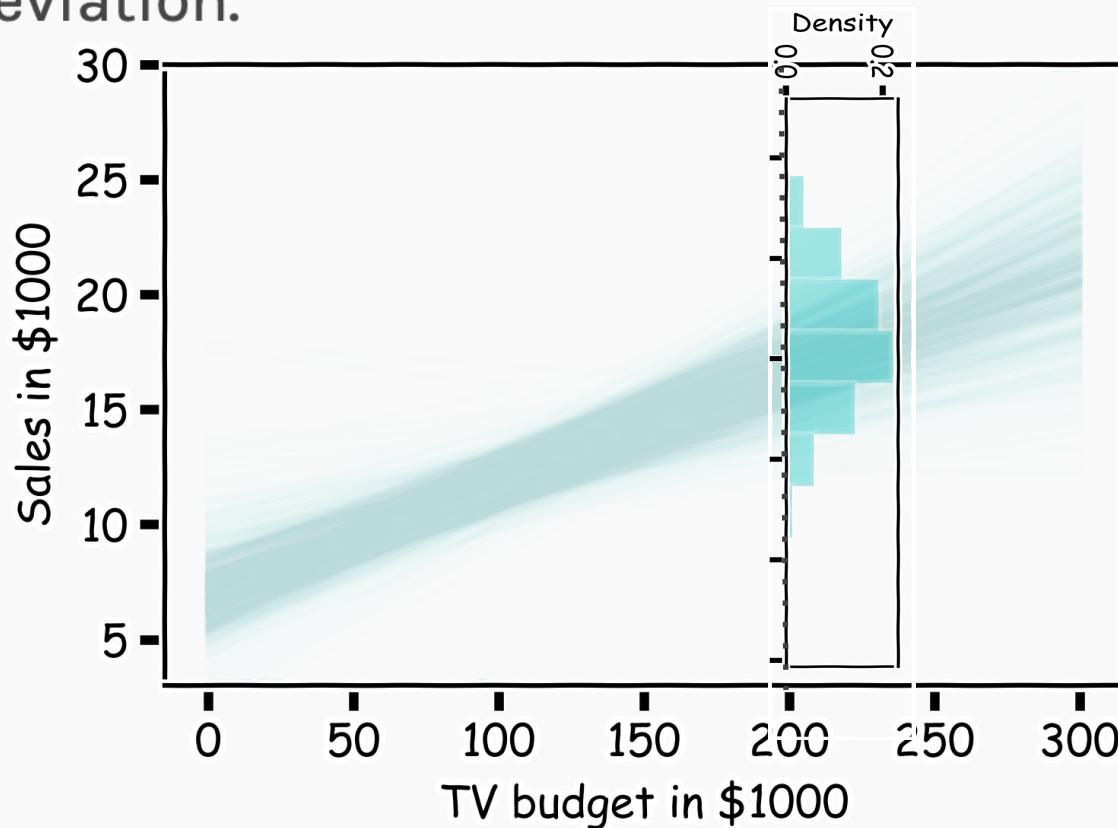
Finally we can calculate the confidence intervals, which are the ranges of values such that the true value of is contained in this interval with n percent probability.



How well do we know \hat{f} ?

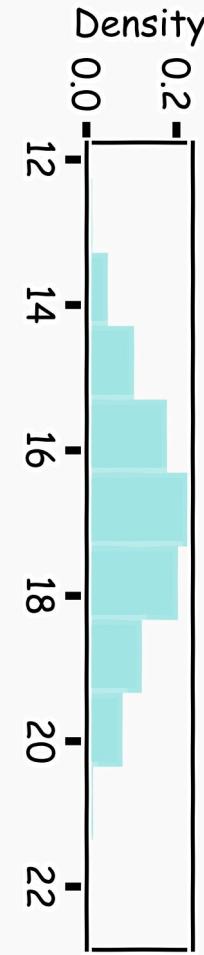
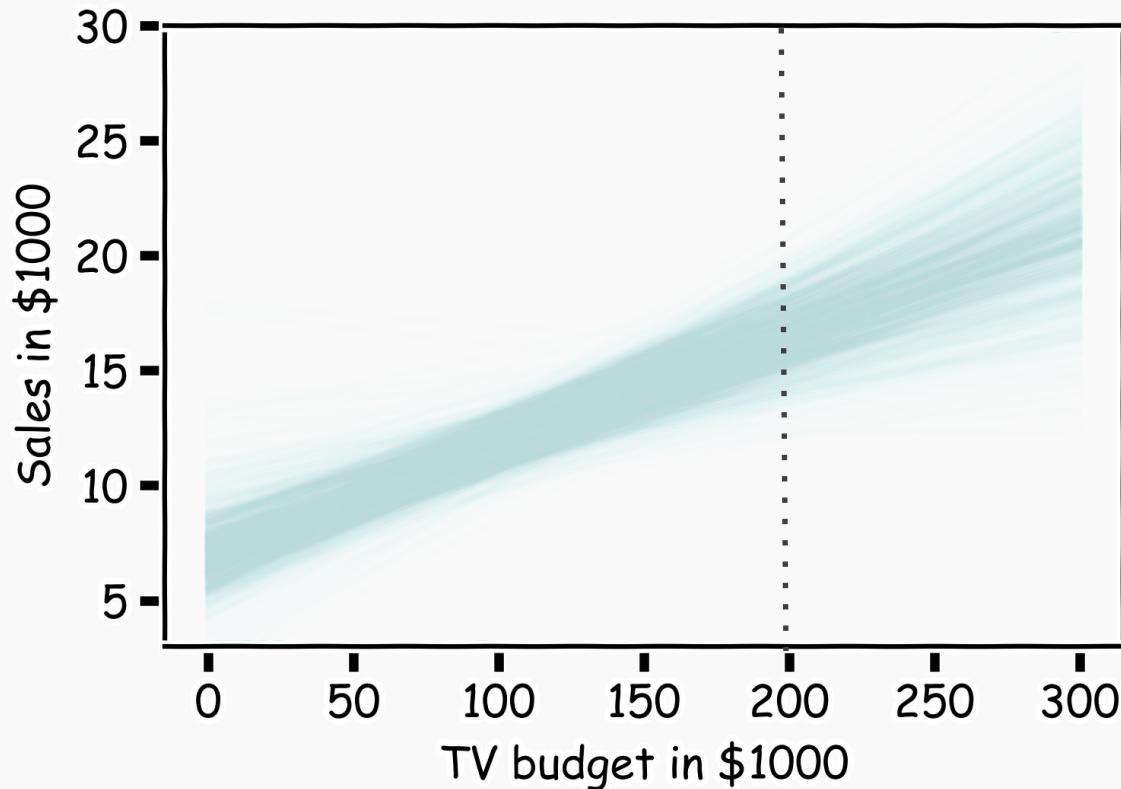
Below we show all regression lines for a thousand of such bootstrapped samples.

For a given x , we examine the distribution of \hat{f} , and determine the mean and standard deviation.



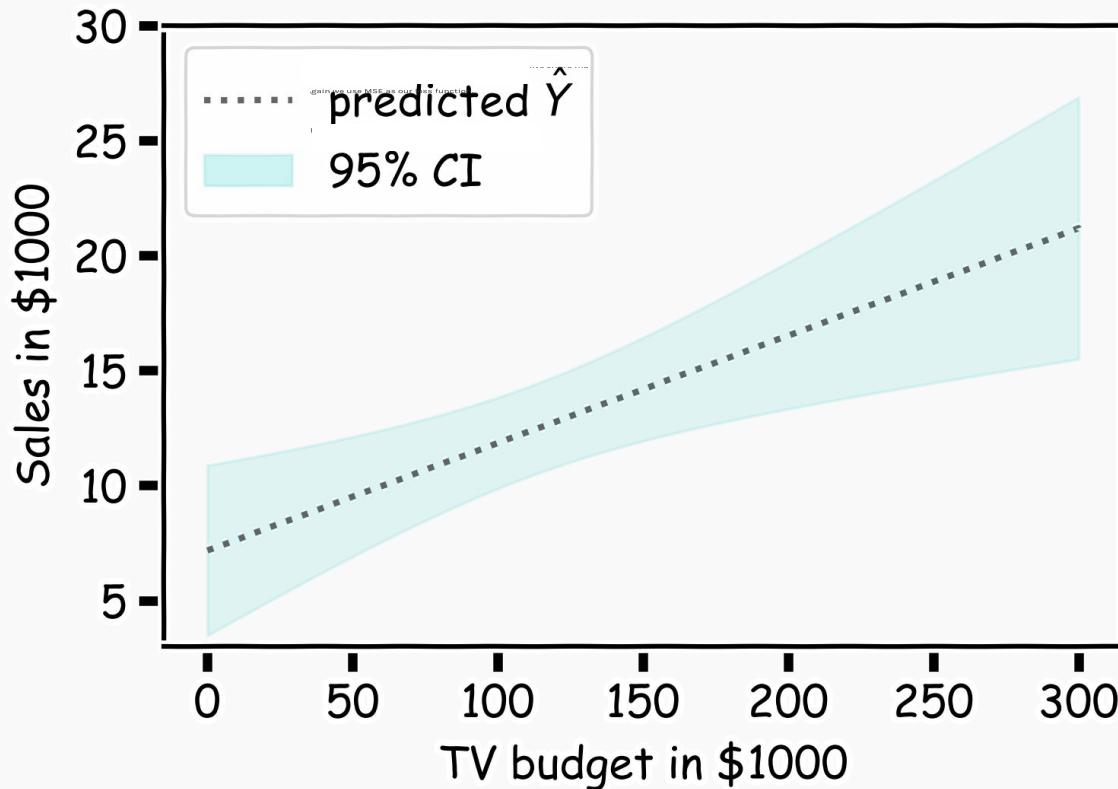
How well do we know \hat{f} ?

Below we show all regression lines for a thousand of such sub-samples. For each one of those “realizations” we can fit a model and test the coefficients.

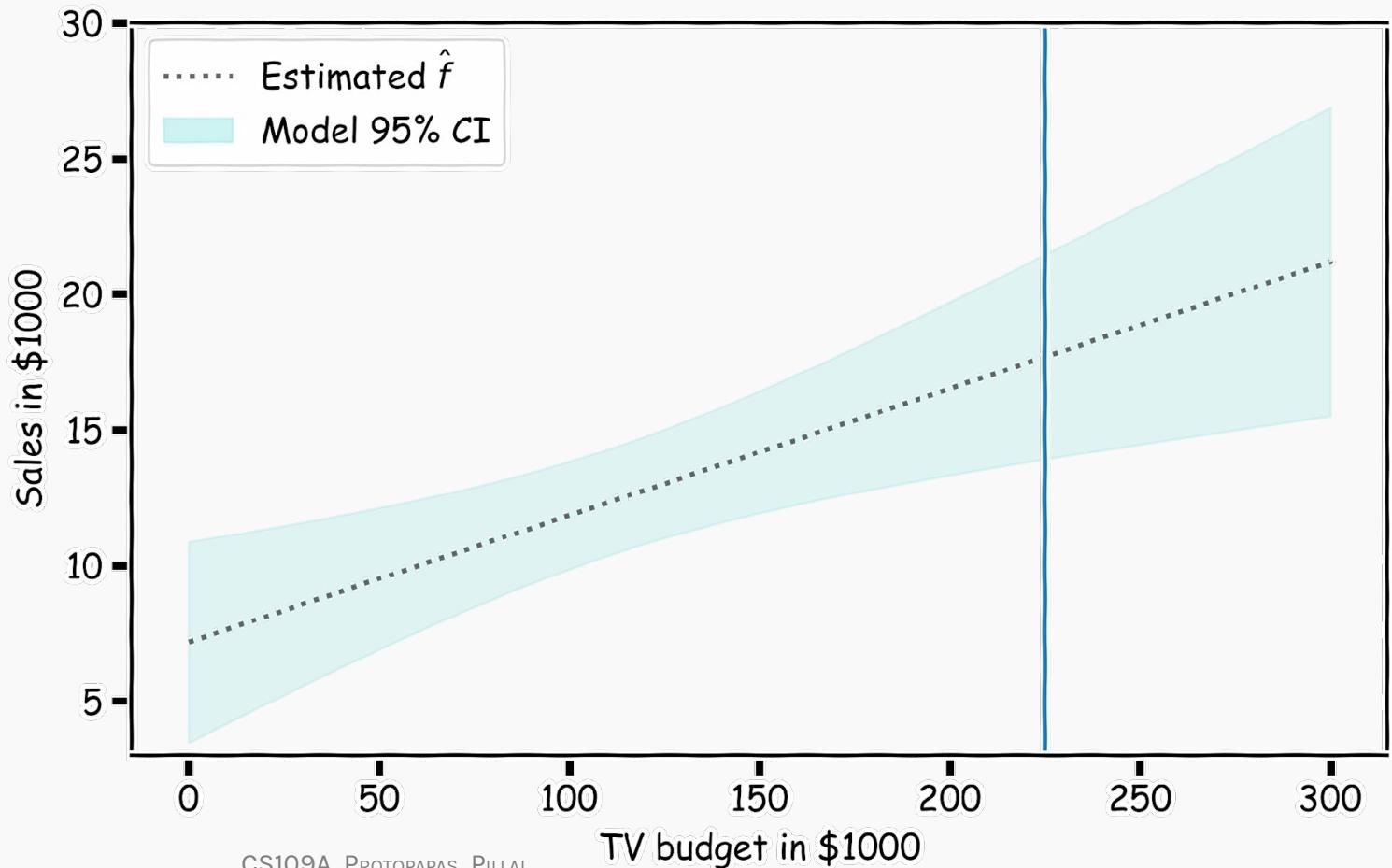


How well do we know \hat{f} ?

For every x , we calculate the mean of the models, \hat{f} (shown with dotted line) and the 95% CI of those models (shaded area).

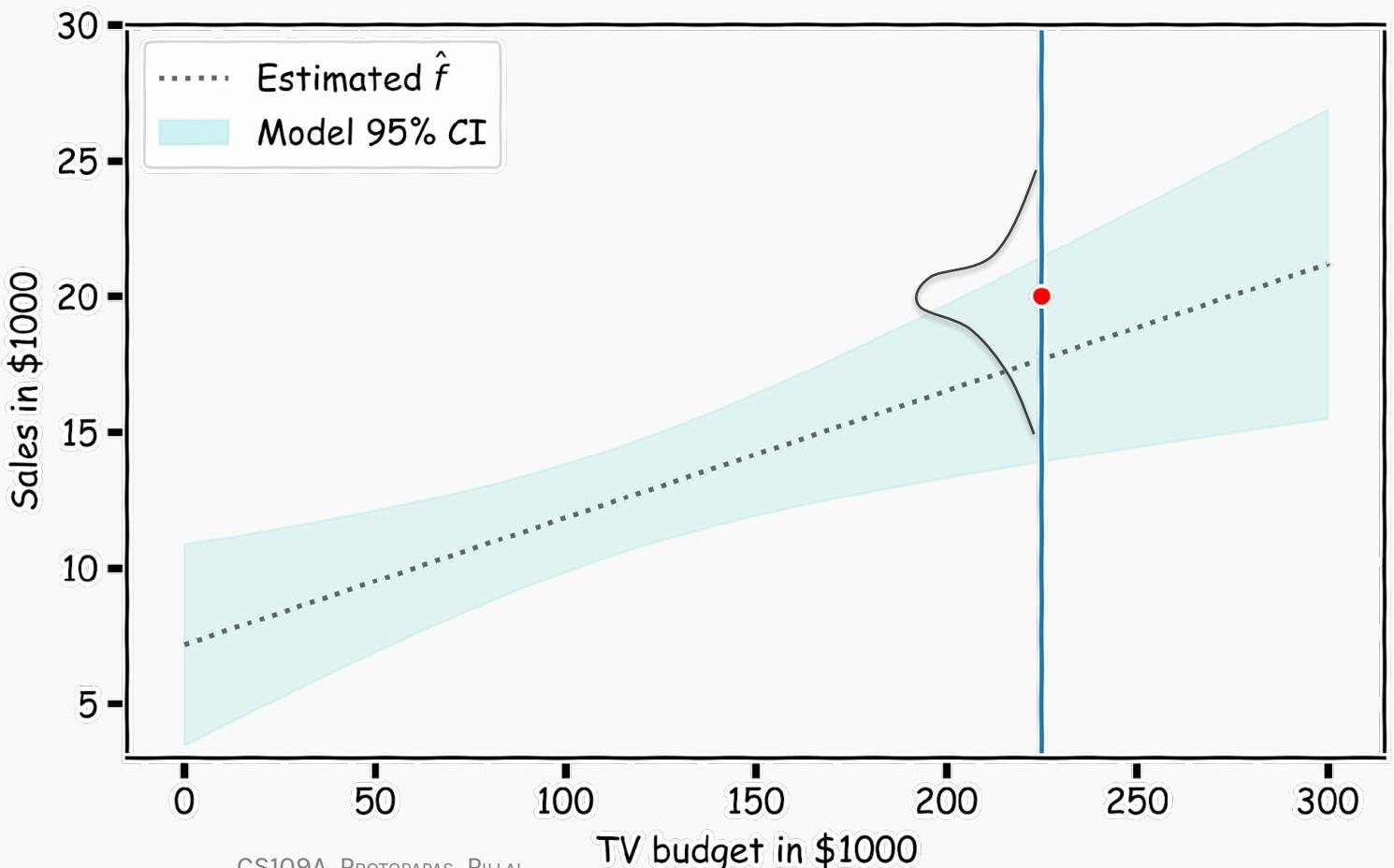


Confidence in predicting \hat{y}



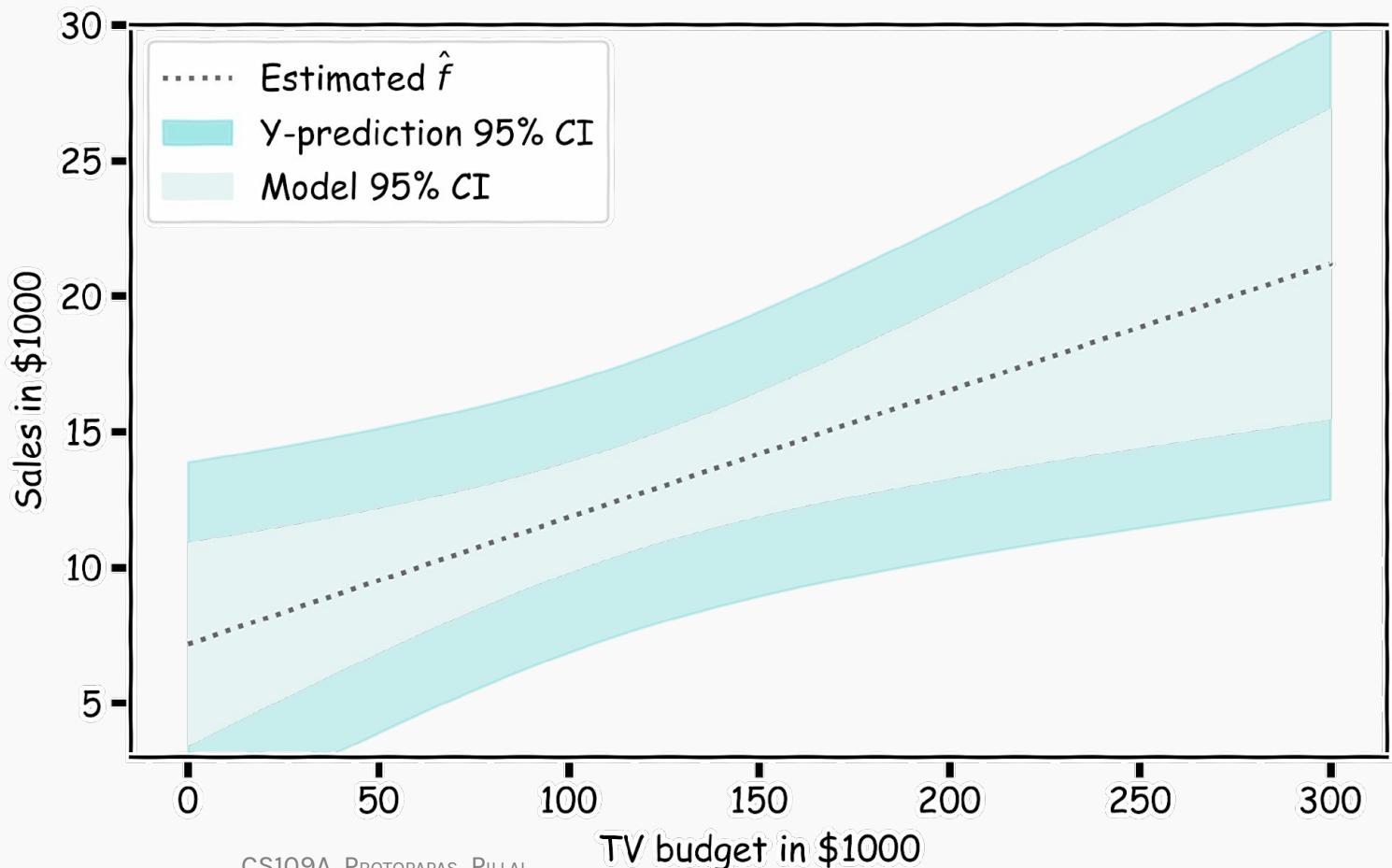
Confidence in predicting \hat{y}

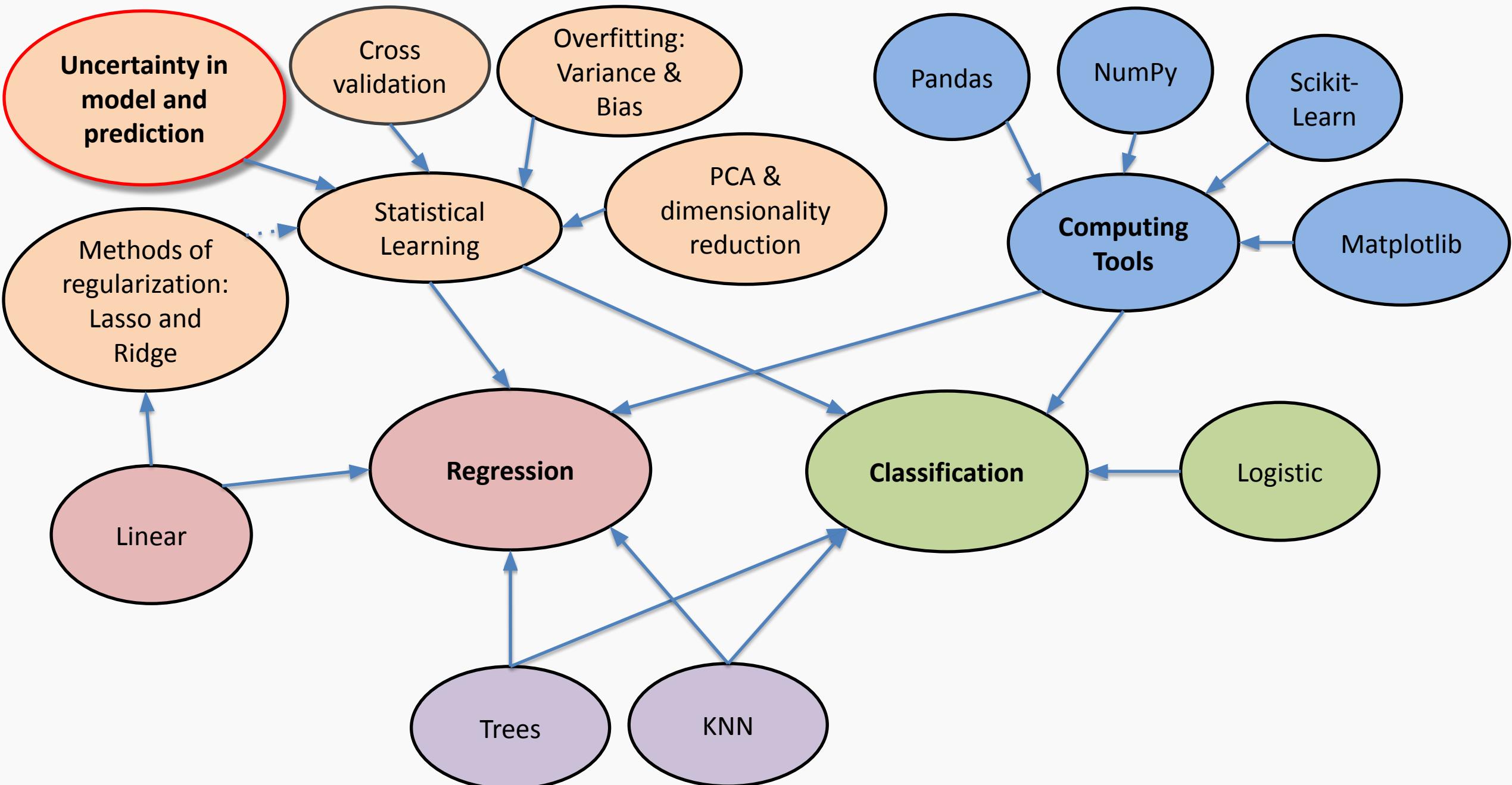
- for a given x , we have a distribution of models
- for each of these the prediction for

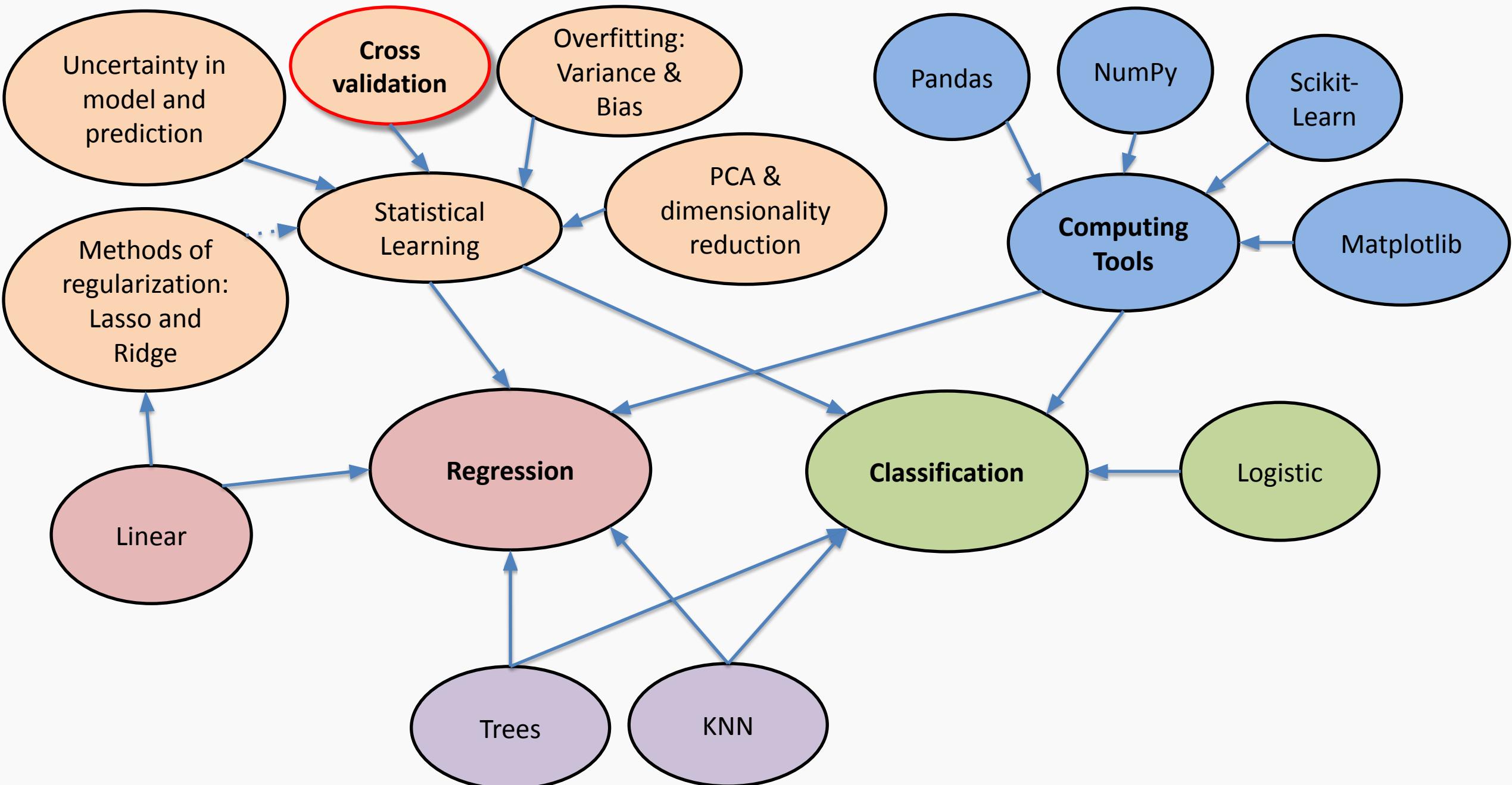


Confidence in predicting \hat{y}

- for a given x , we have a distribution of models $f(x)$
- for each of these $f(x)$, the prediction for $y \sim N(f, \sigma_e)$

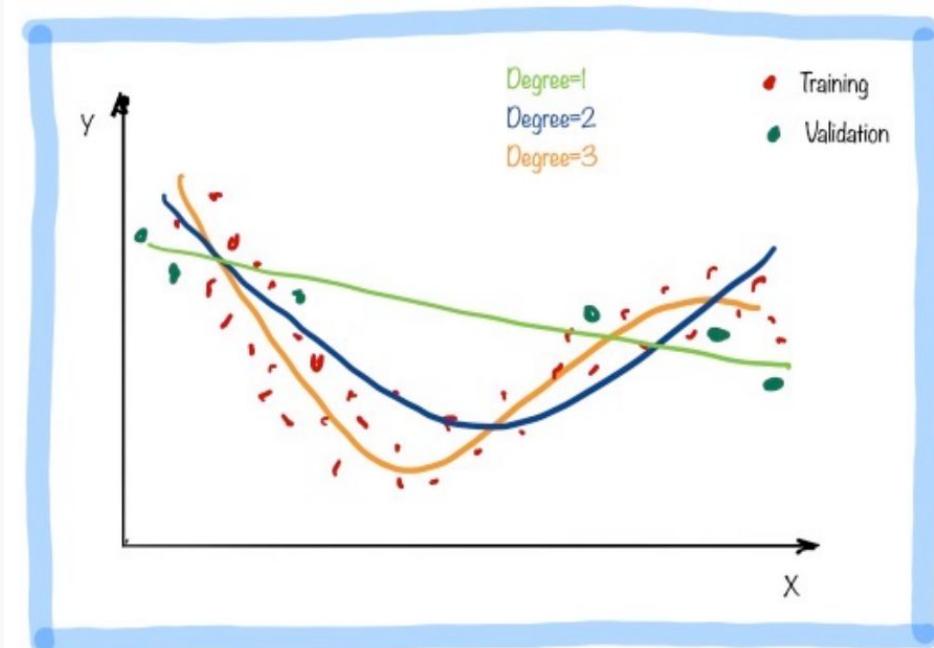






Cross Validation: Motivation

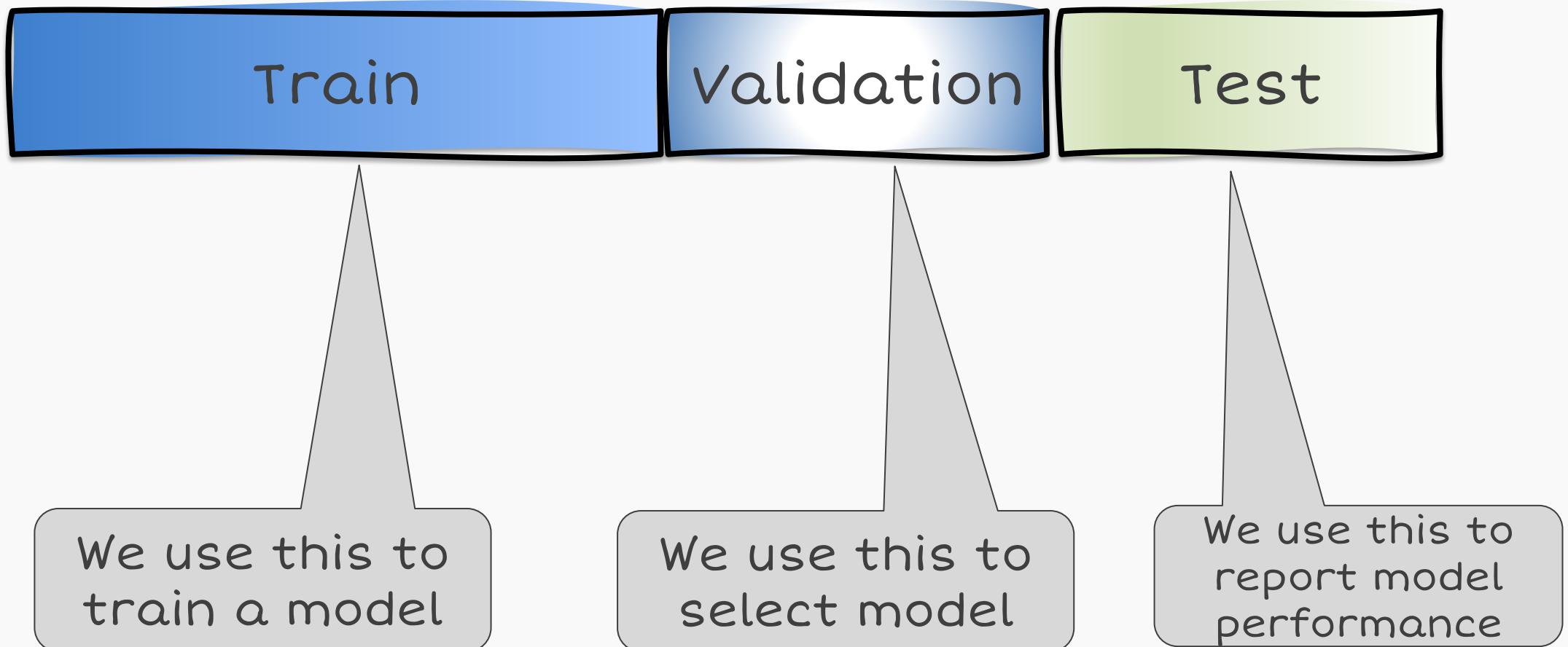
Using a single validation set to select amongst multiple models can be problematic - **there is the possibility of overfitting to the validation set.**



It is obvious that degree=3 is the correct model but the validation set by chance favors the linear model.

Train-Validation-Test

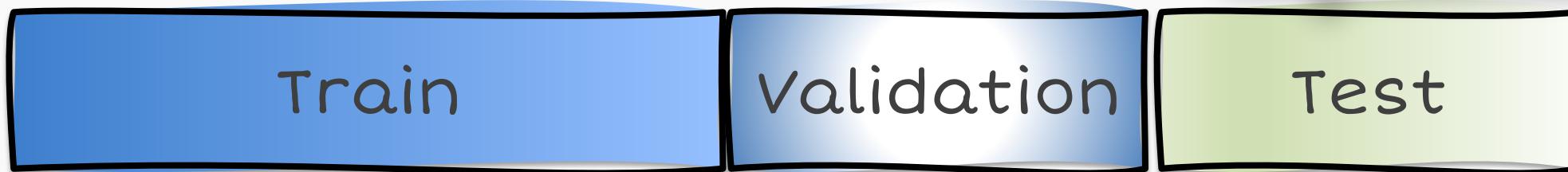
We introduce a different sub-set, which we called validation and we use it to select the model.



Train-Validation-Test

We introduce a different sub-set, which we called validation to select the model.

The test set should never be touched for model training or selection.

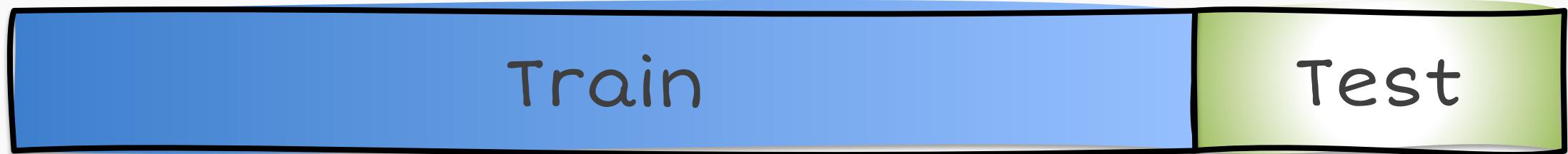


We use this to train a model

We use this to select model

We use this to report model performance

Cross Validation



Validate Model

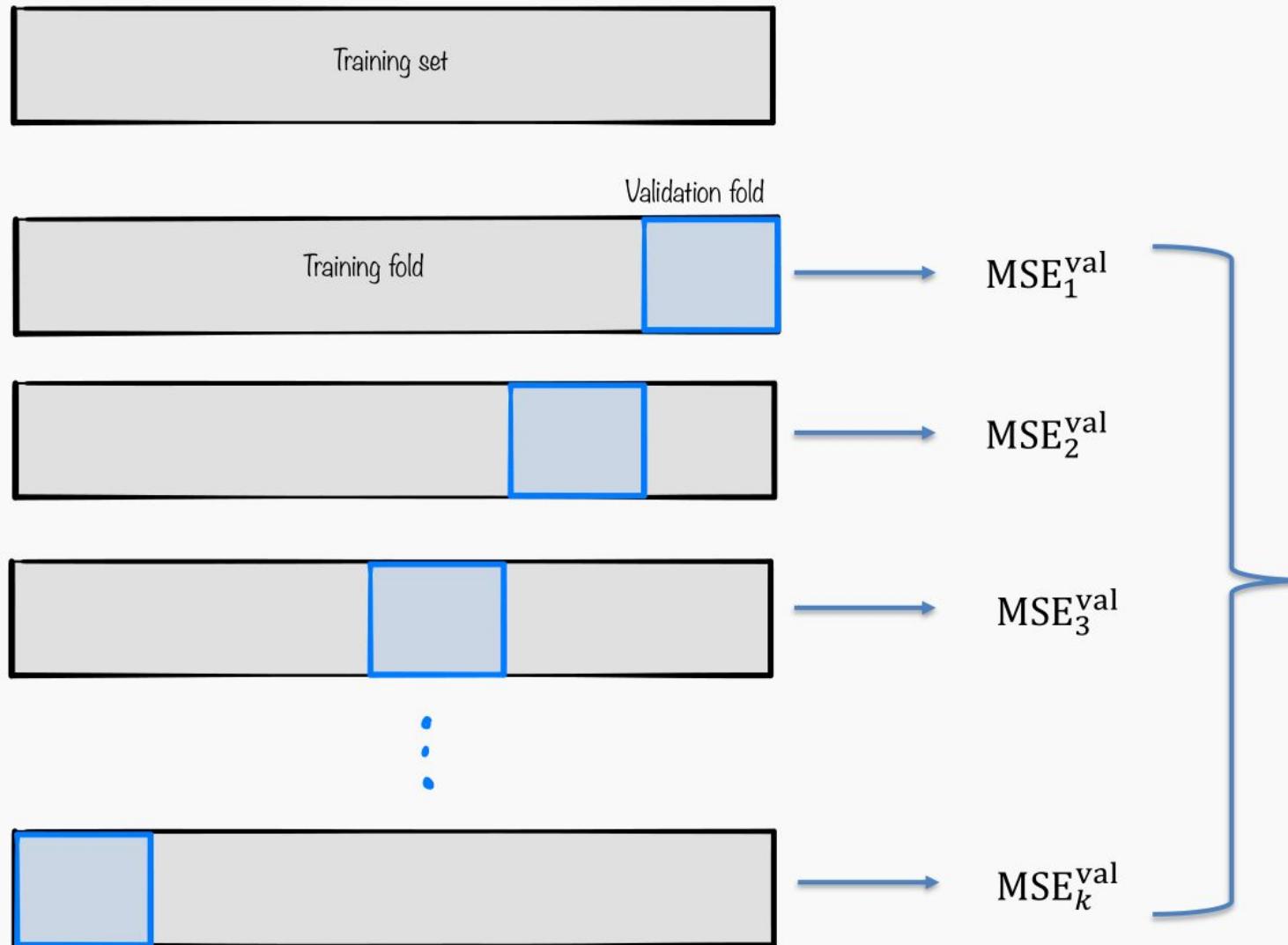
$$MSE_5^{val}$$

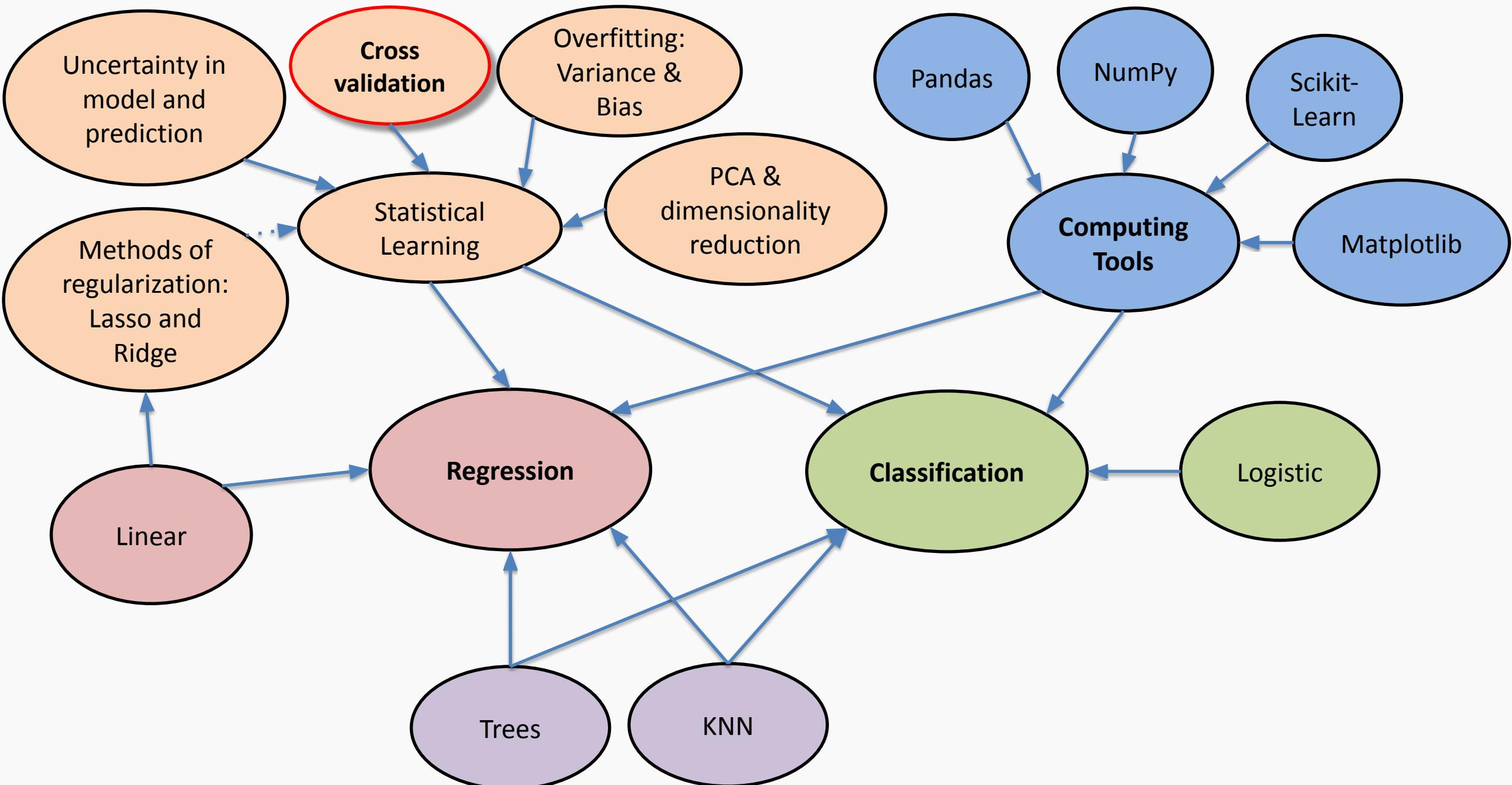
Train
model

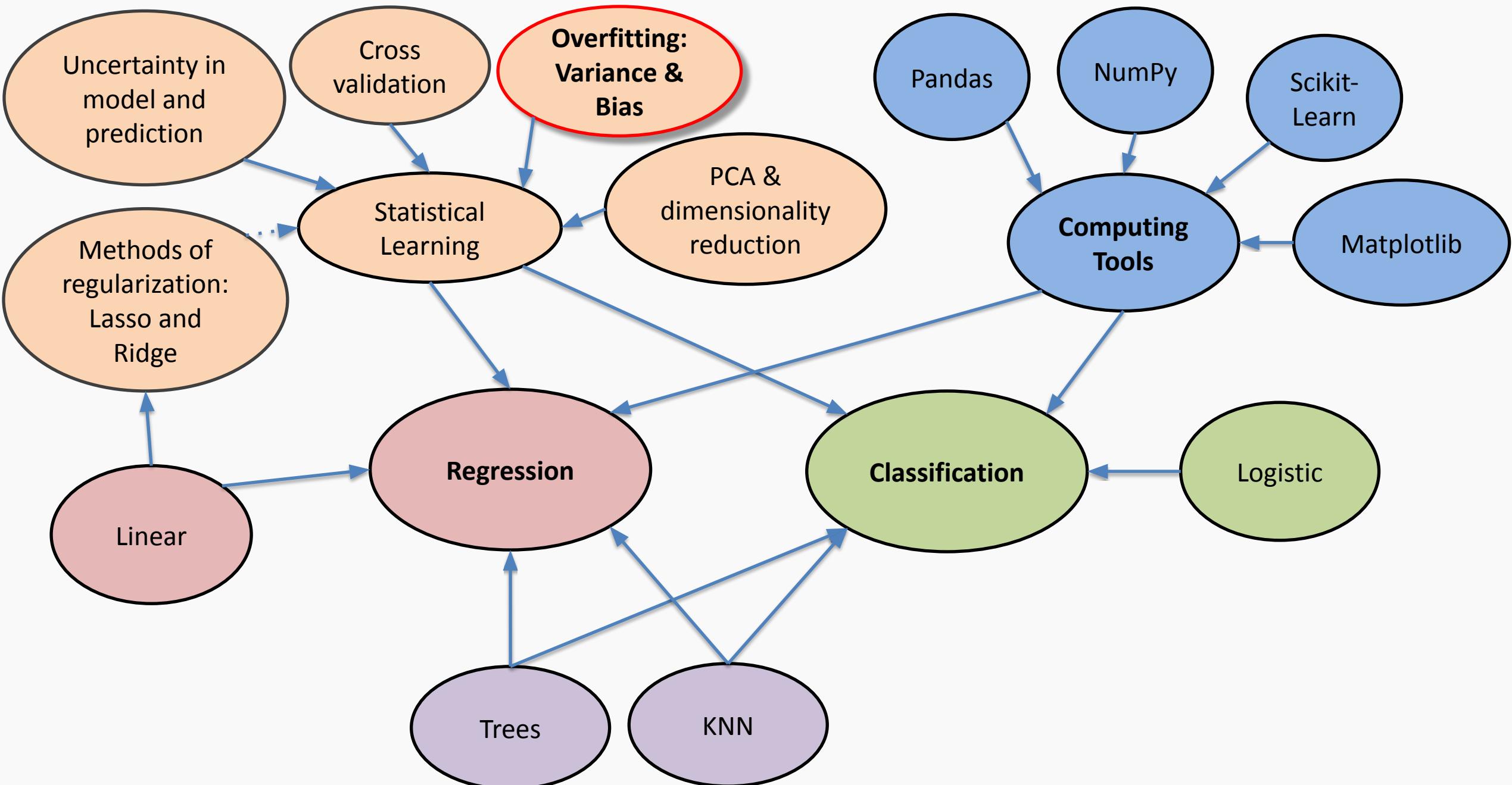
$$MSE^{val} = \frac{1}{5} \sum_{i=1}^5 MSE_i^{val}$$



Cross Validation



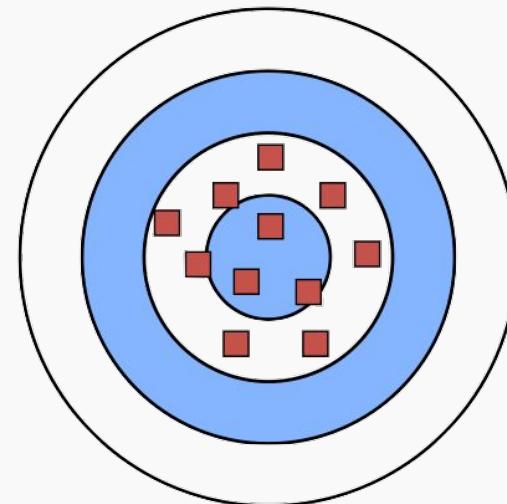
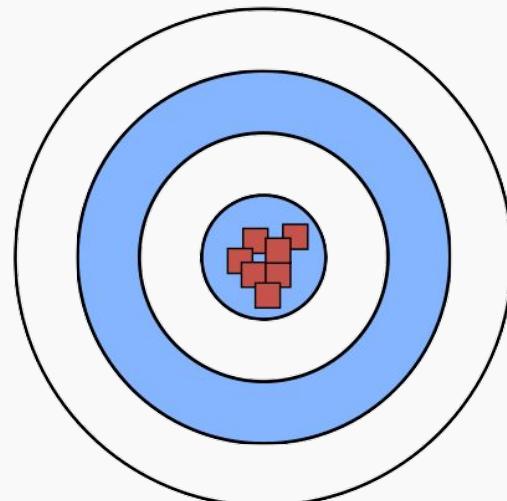




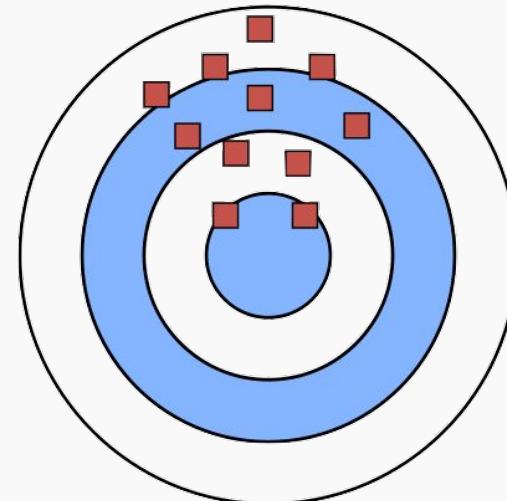
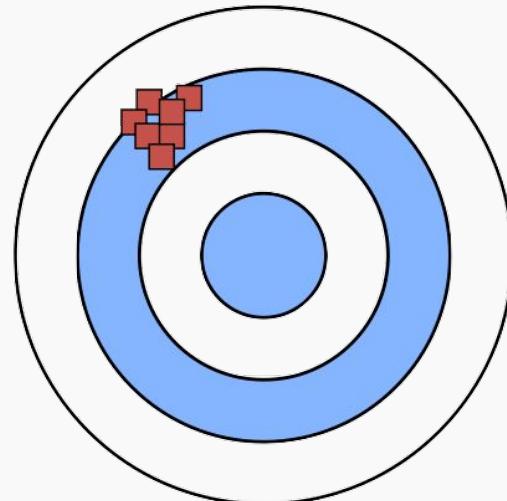
Low Variance
(Precise)

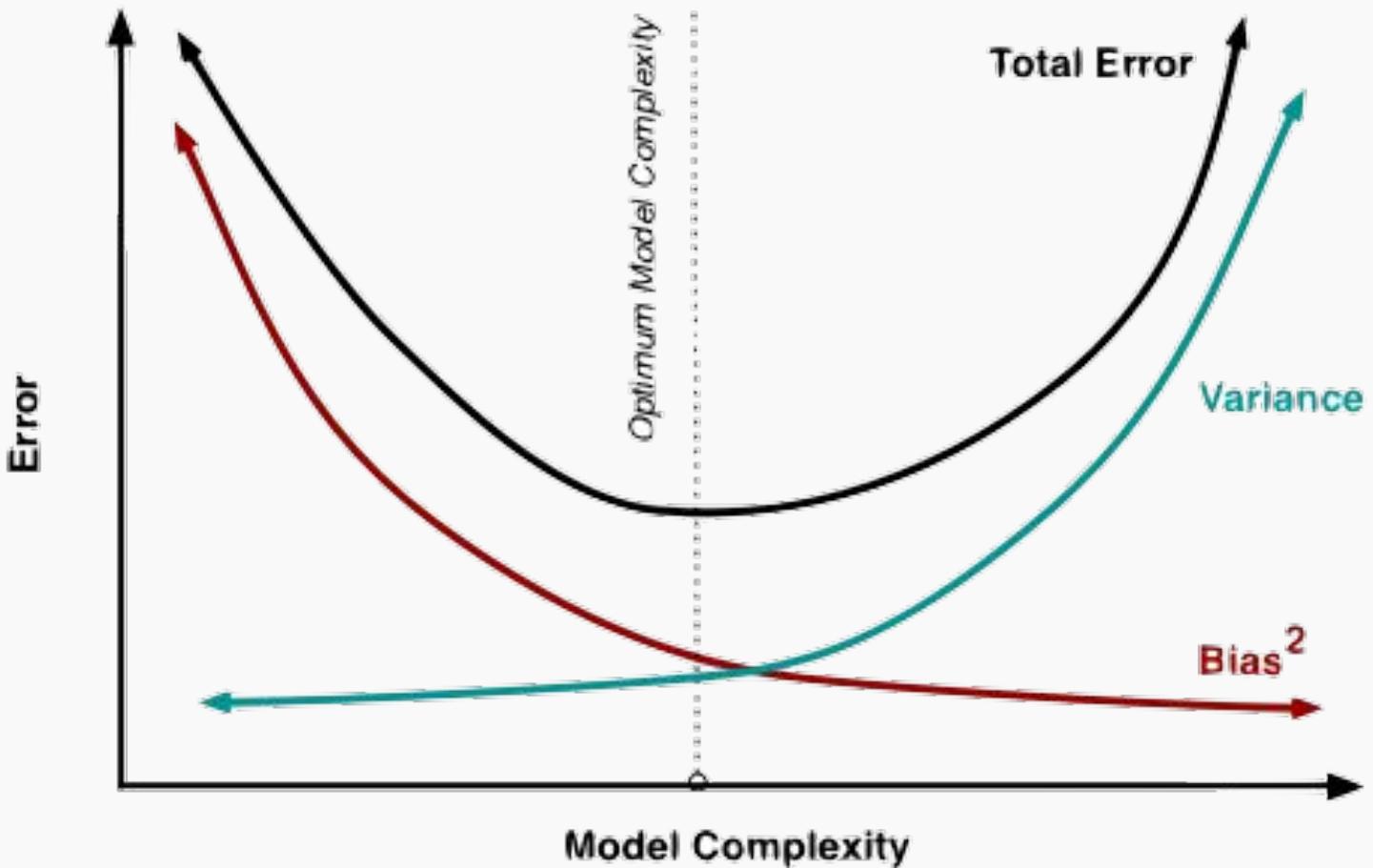
High Variance
(Not Precise)

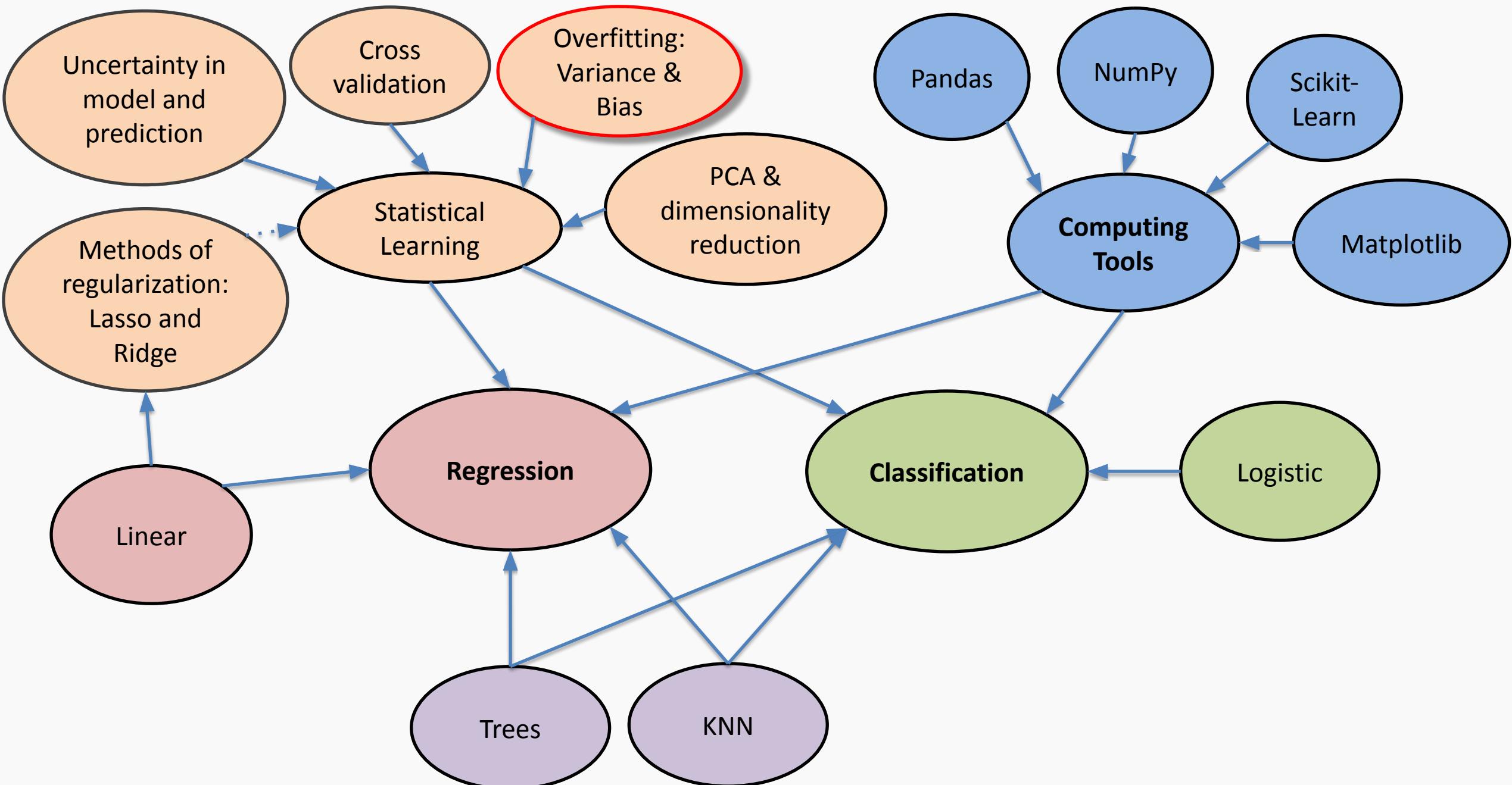
Low Bias
(Accurate)

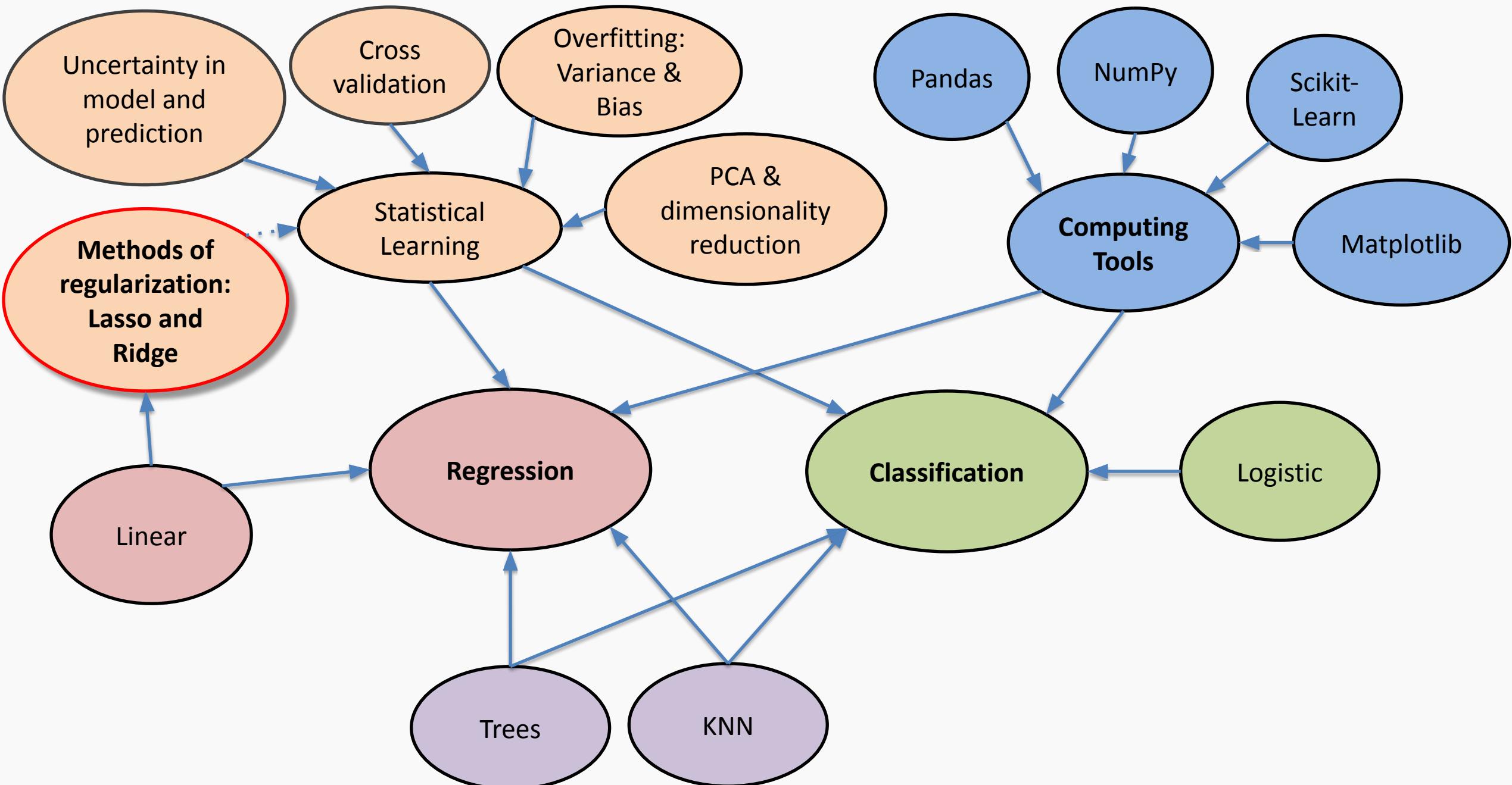


High Bias
(Not Accurate)

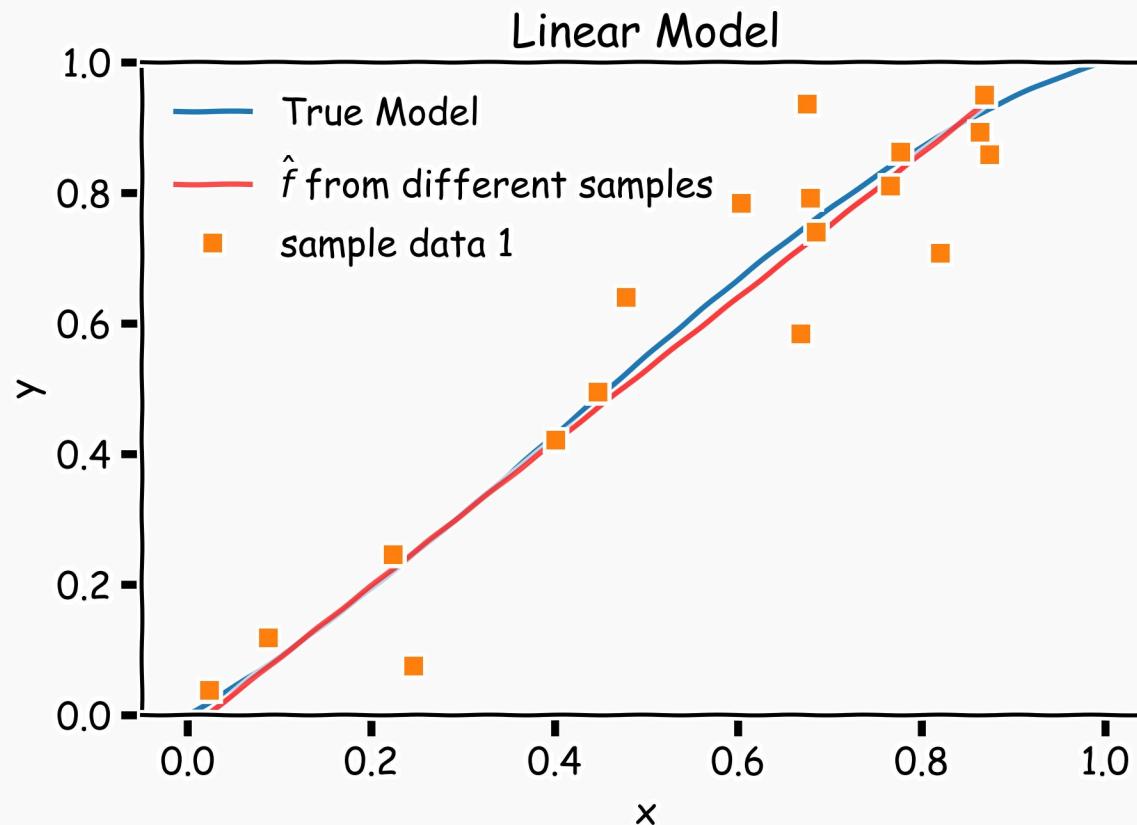




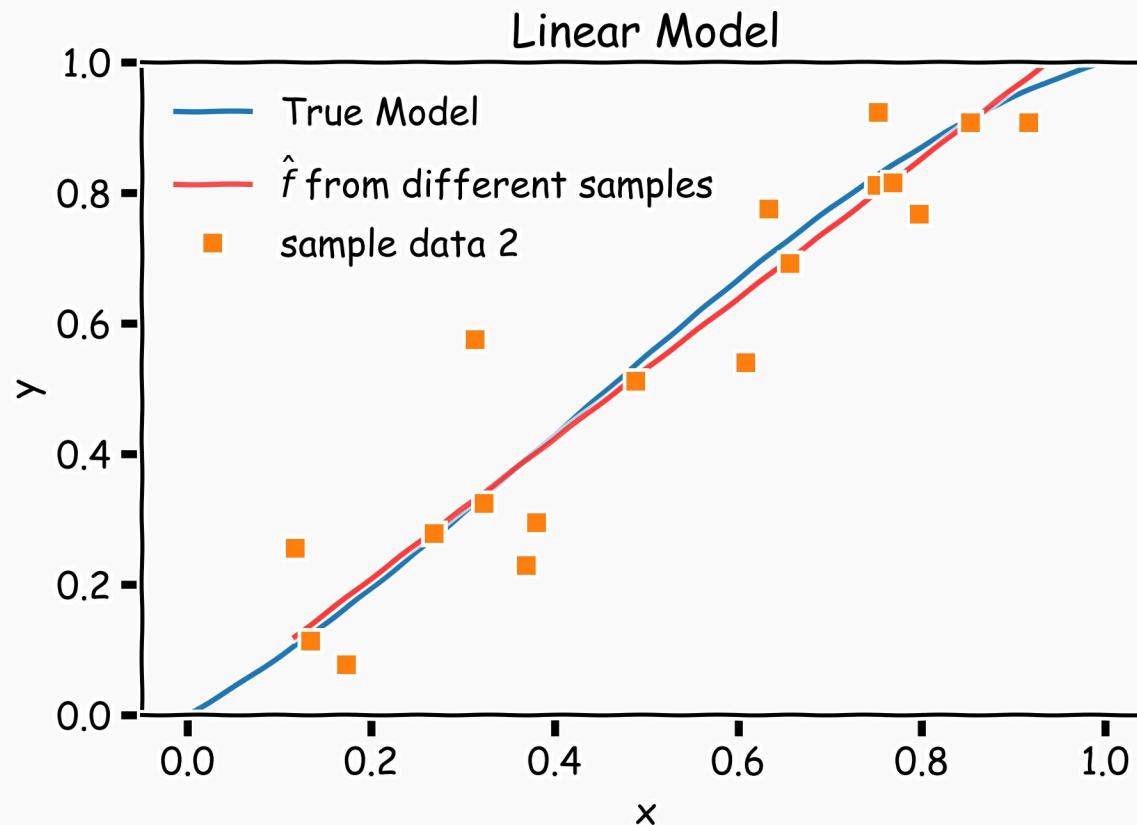




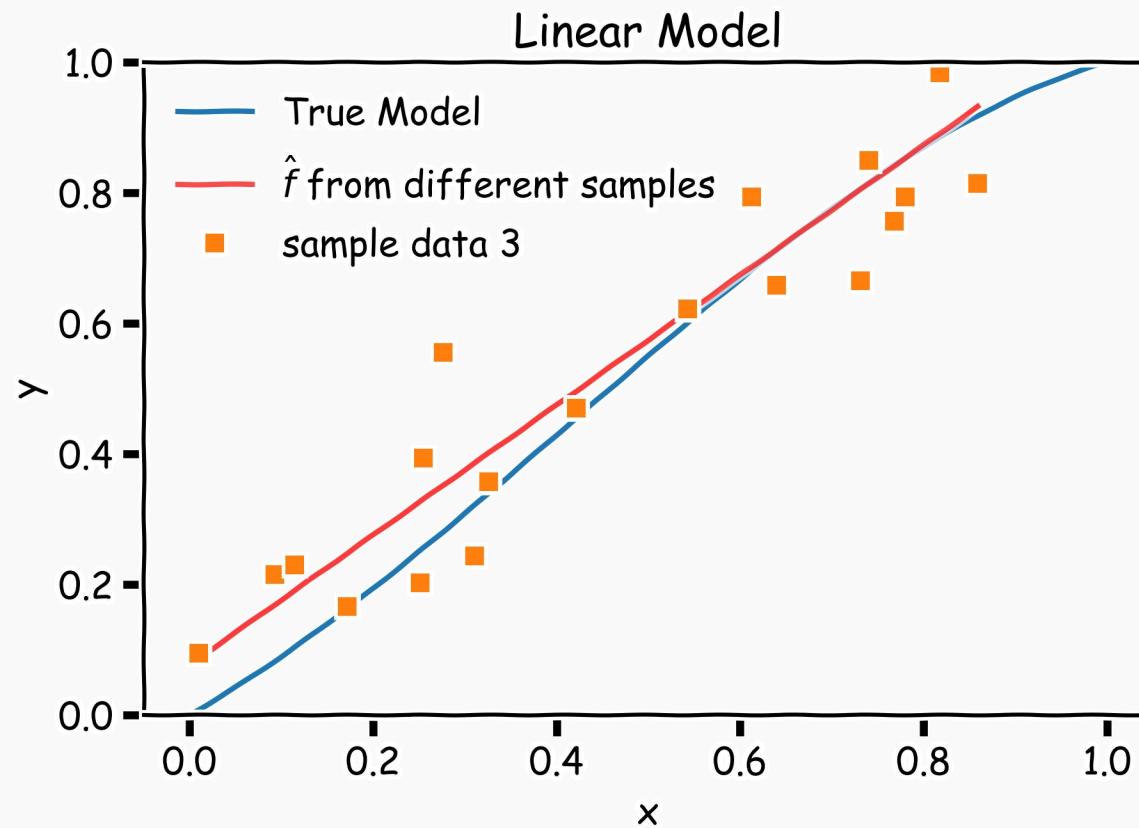
Bias vs Variance



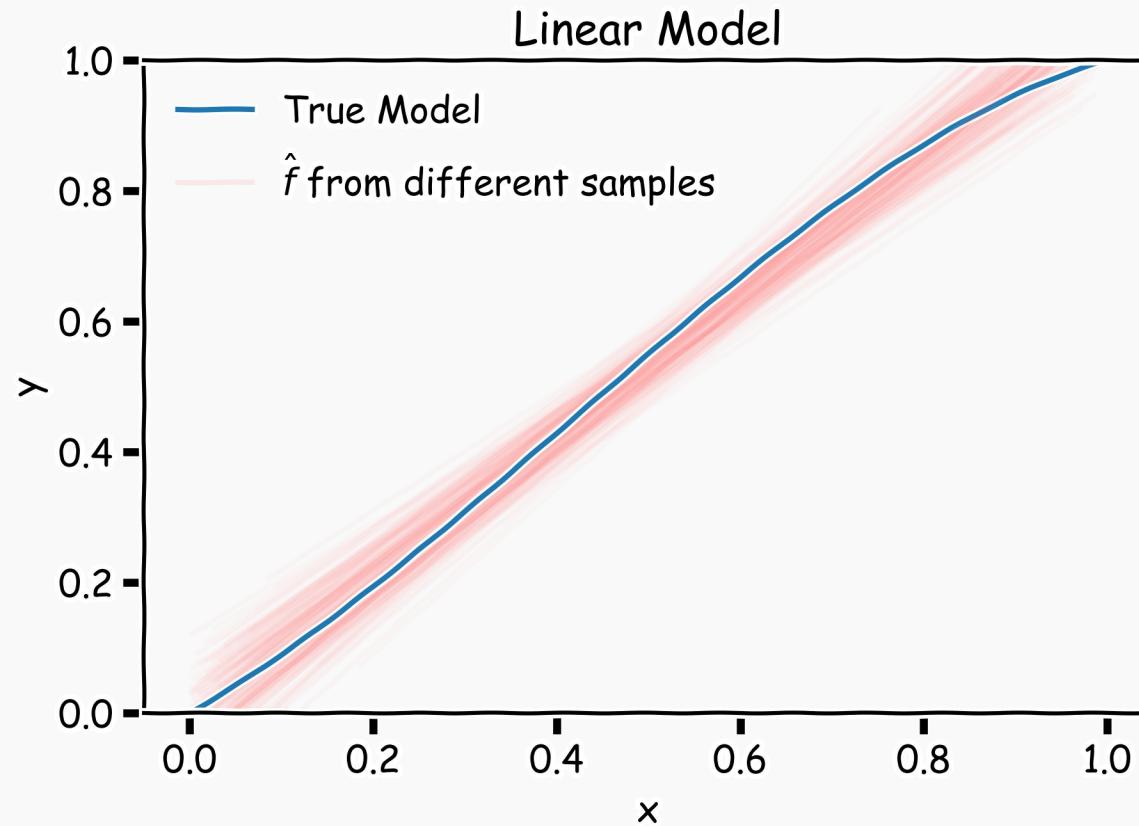
Bias vs Variance



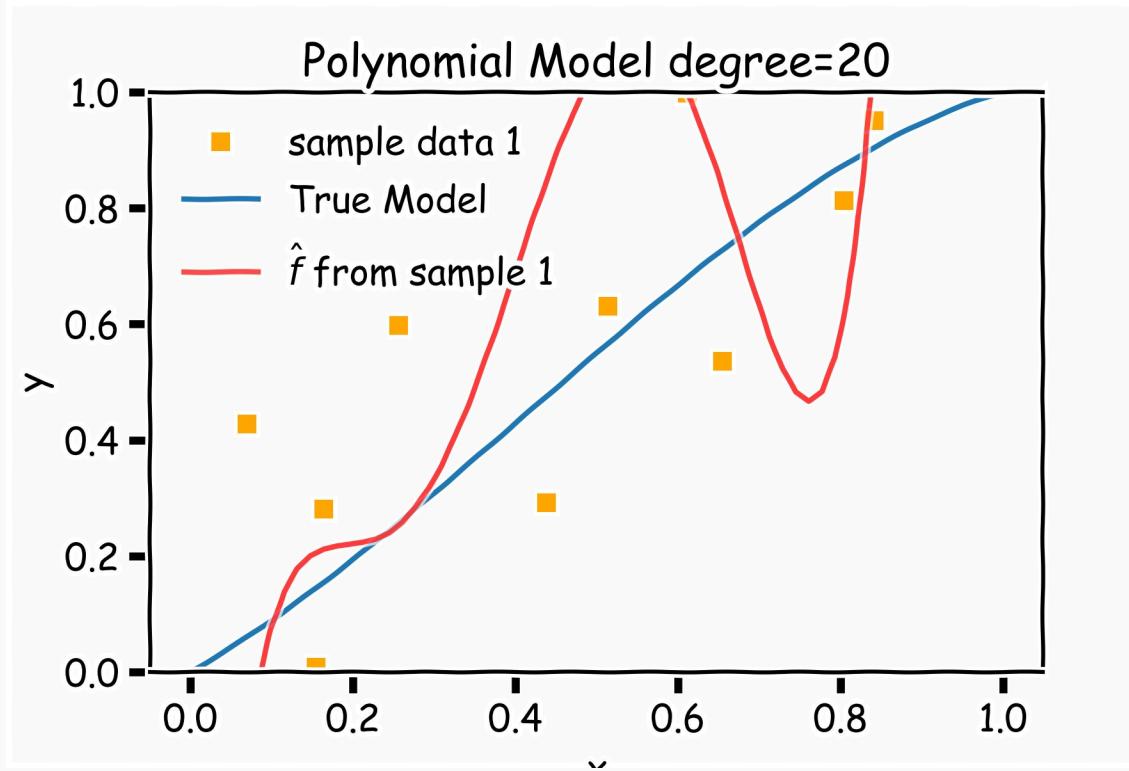
Bias vs Variance



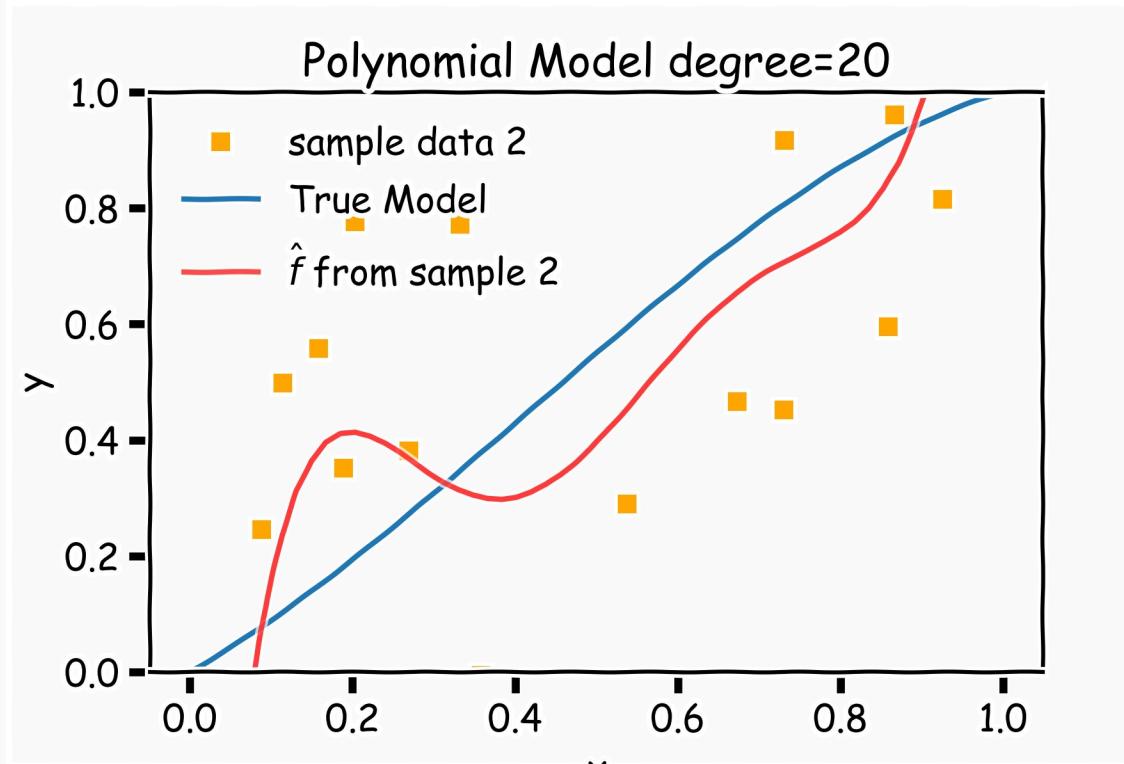
Linear models: 20 data points per line 2000 simulations



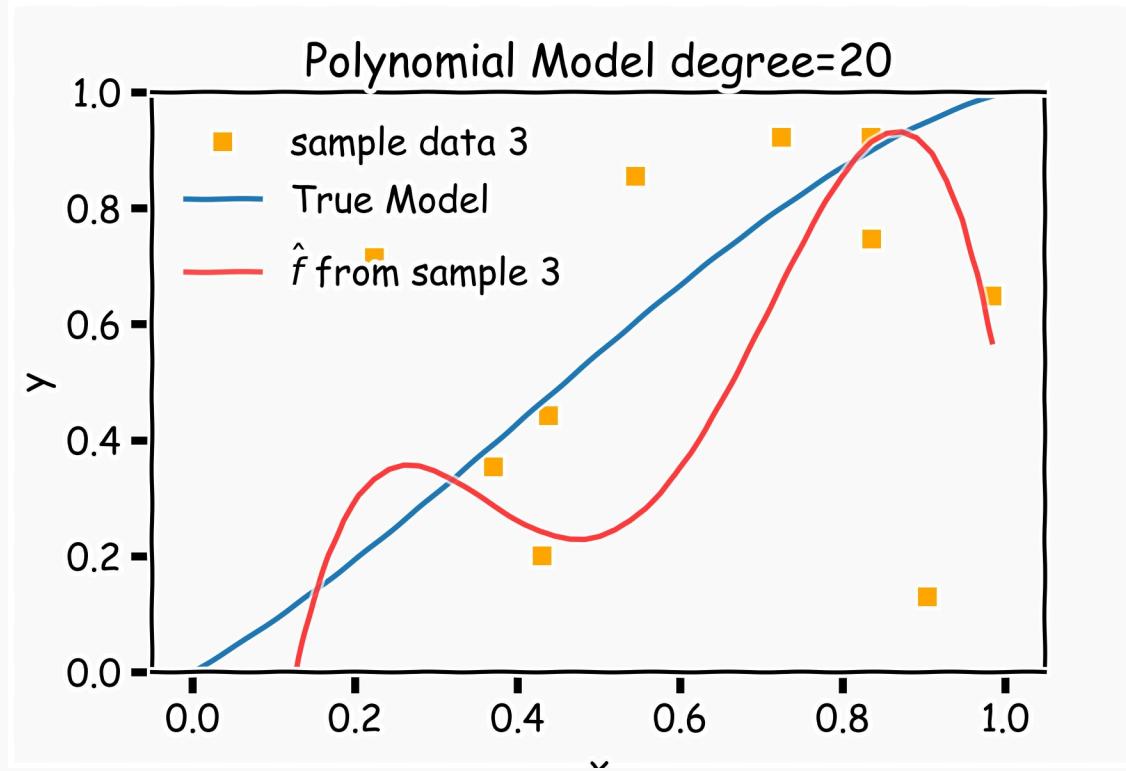
Bias vs Variance



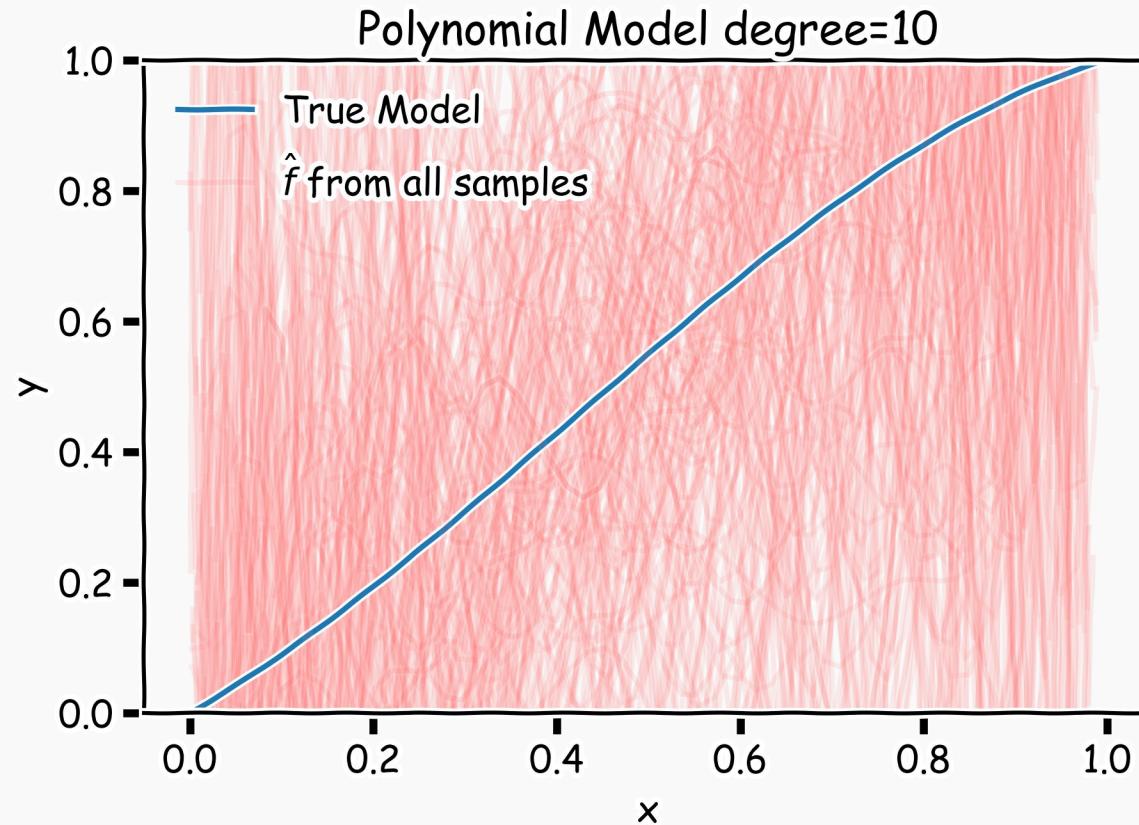
Bias vs Variance



Bias vs Variance



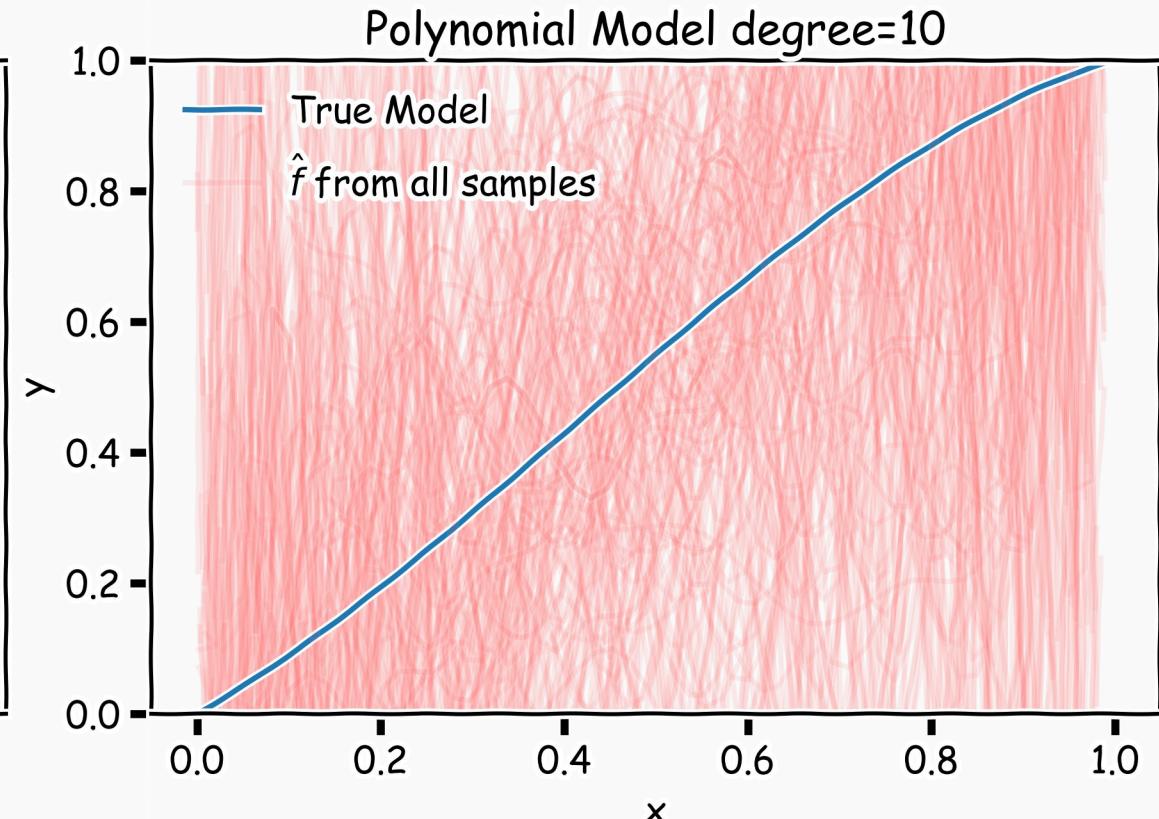
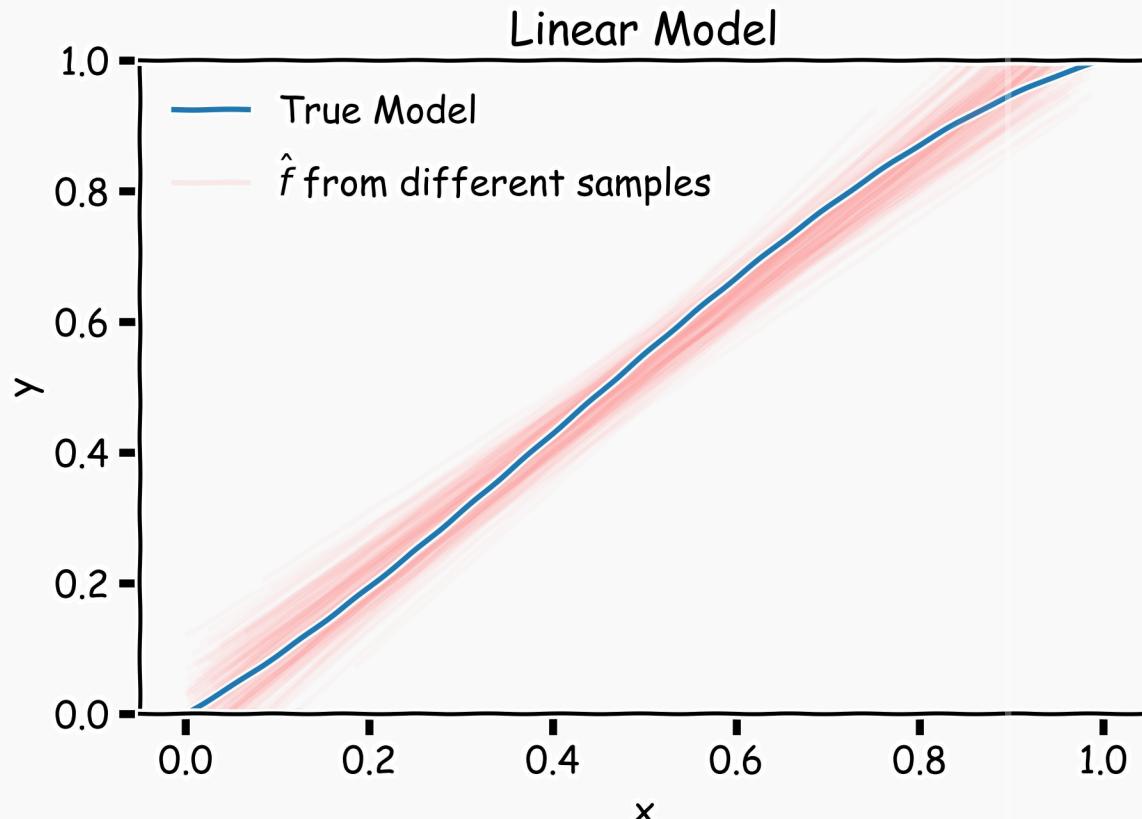
Poly 10 degree models : 20 data points per line 2000 simulations



Bias vs Variance

Left: 2000 best fit straight lines, each fitted on a different 20 point training set.

Right: Best-fit models using degree 10 polynomial



Regularization: An Overview

The idea of regularization revolves around modifying the loss function L ; in particular, we add a regularization term that penalizes some specified properties of the model parameters

$$L_{reg}(\beta) = L(\beta) + \lambda R(\beta),$$

LASSO Regression

Since we wish to discourage extreme values in model parameter, we need to choose a regularization term that penalizes parameter magnitudes. For our loss function, we will again use MSE.

Together our regularized loss function is:

$$L_{LASSO}(\beta) = \frac{1}{n} \sum_{i=1}^n |y_i - \beta^\top \mathbf{x}_i|^2 + \lambda \sum_{j=1}^J |\beta_j|.$$

Note that $\sum_{j=1}^J |\beta_j|$ is the l_1 norm of the vector β

$$\sum_{j=1}^J |\beta_j| = \|\beta\|_1$$

Ridge Regression

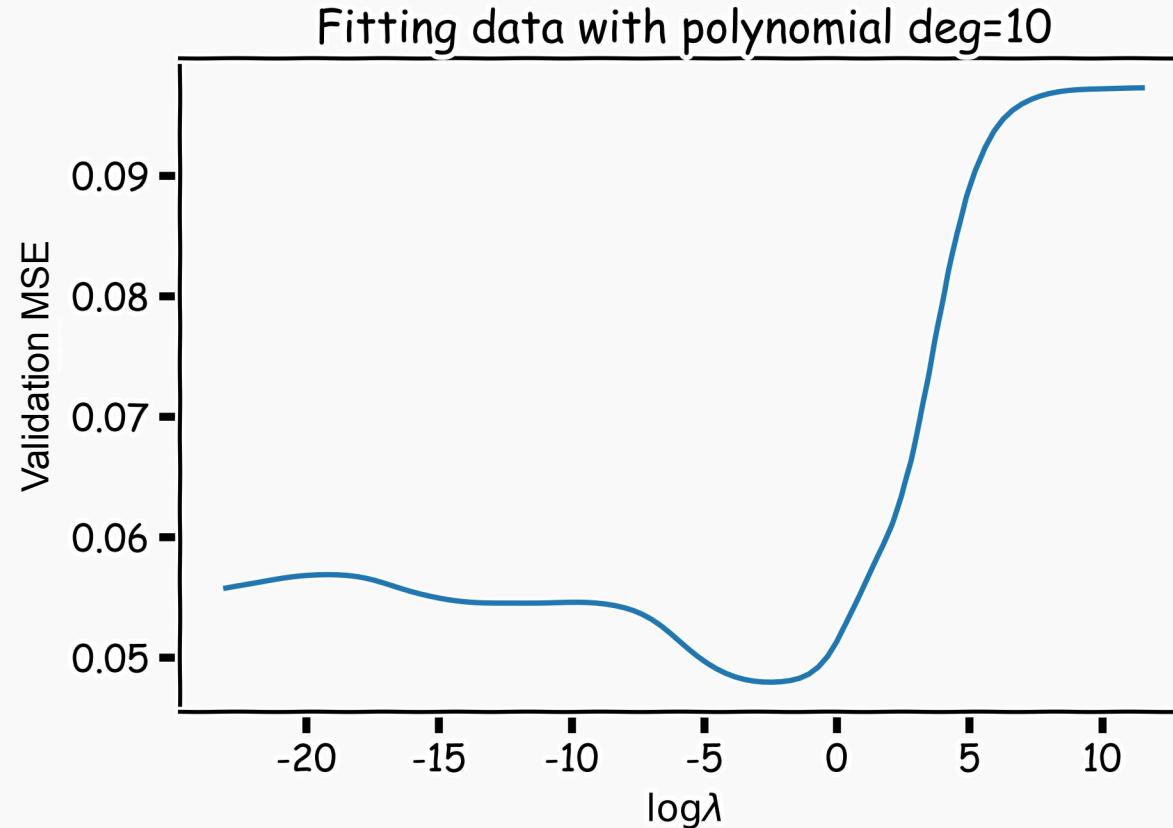
Alternatively, we can choose a regularization term that penalizes the squares of the parameter magnitudes. Then, our regularized loss function is:

$$L_{Ridge}(\beta) = \frac{1}{n} \sum_{i=1}^n |y_i - \boldsymbol{\beta}^\top \mathbf{x}_i|^2 + \lambda \sum_{j=1}^J \beta_j^2.$$

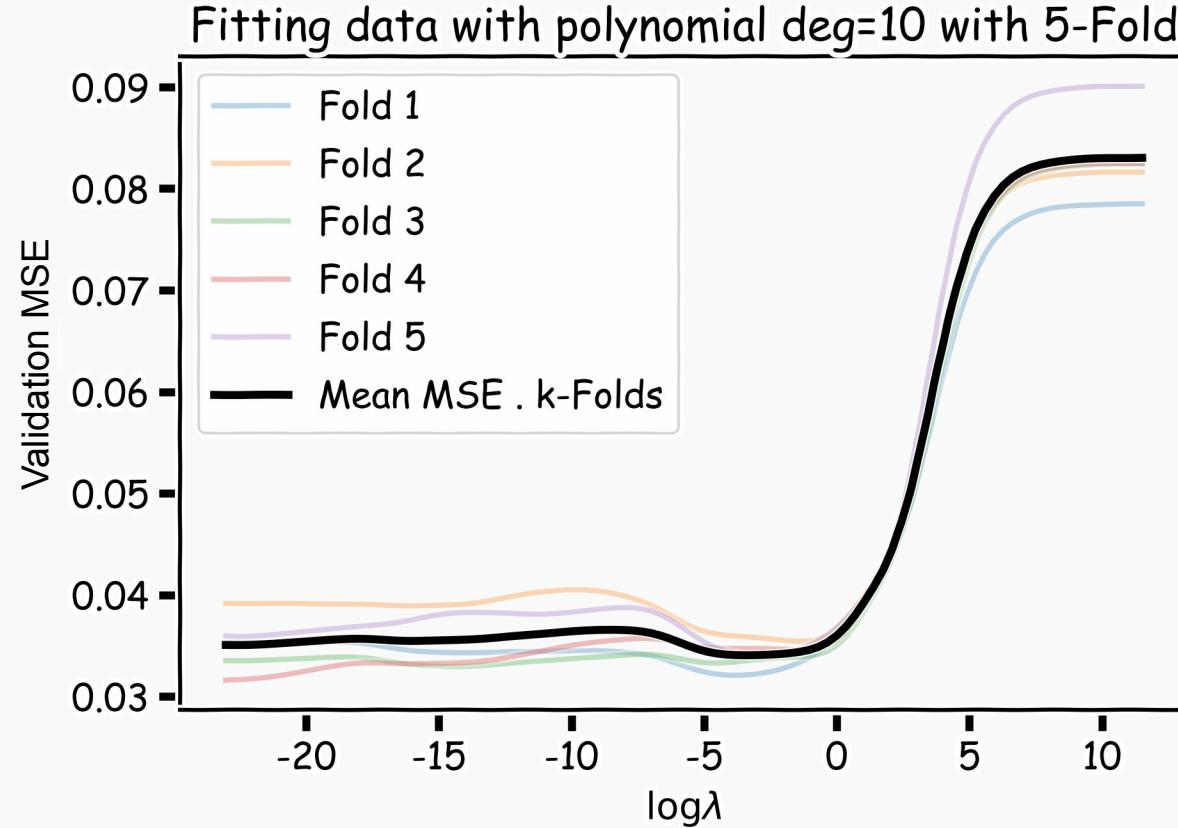
Note that $\sum_{j=1}^J \beta_j^2$ is the L_2 norm of the vector $\boldsymbol{\beta}$

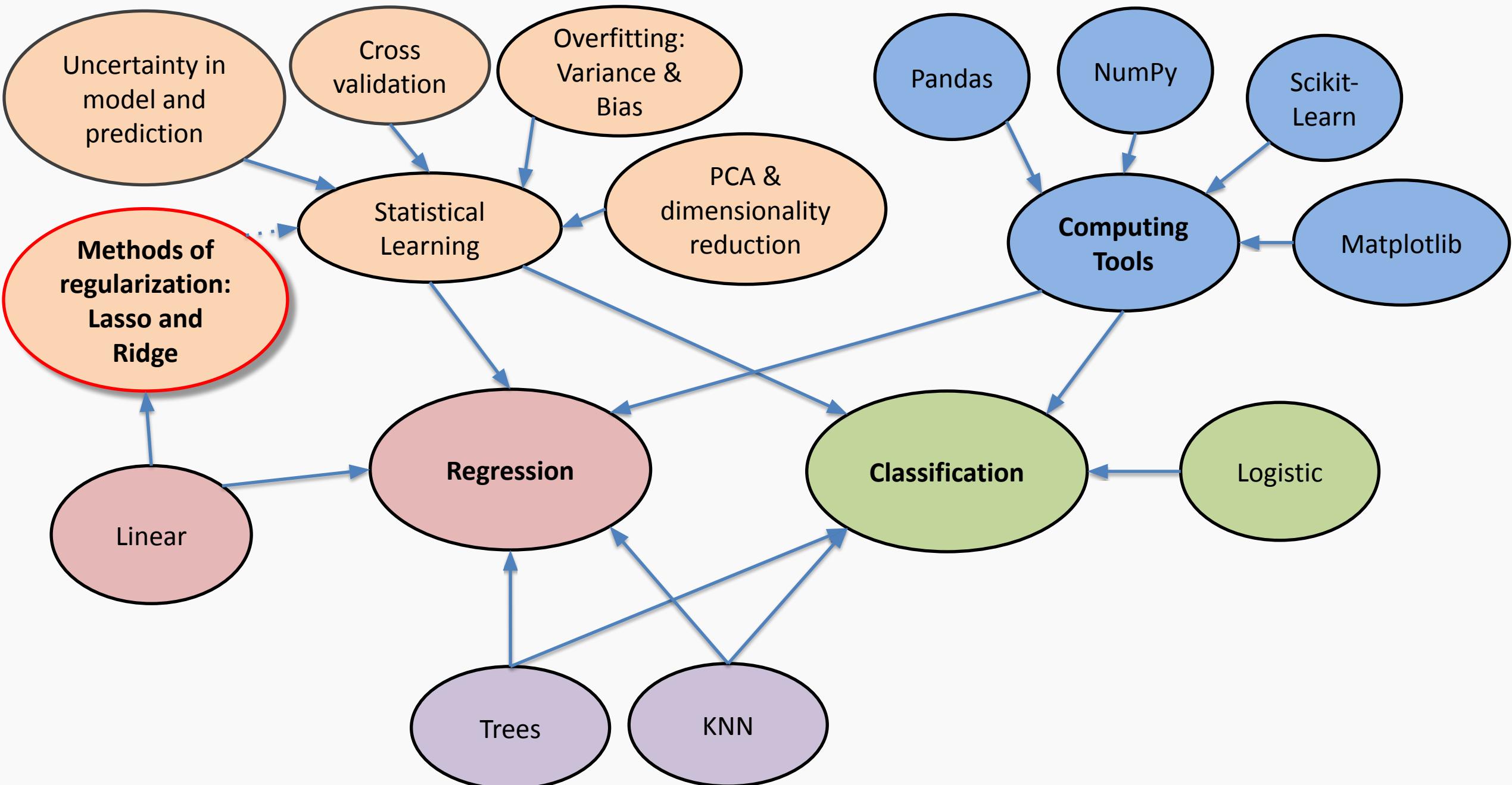
$$\sum_{j=1}^J \beta_j^2 = \|\boldsymbol{\beta}\|_2^2$$

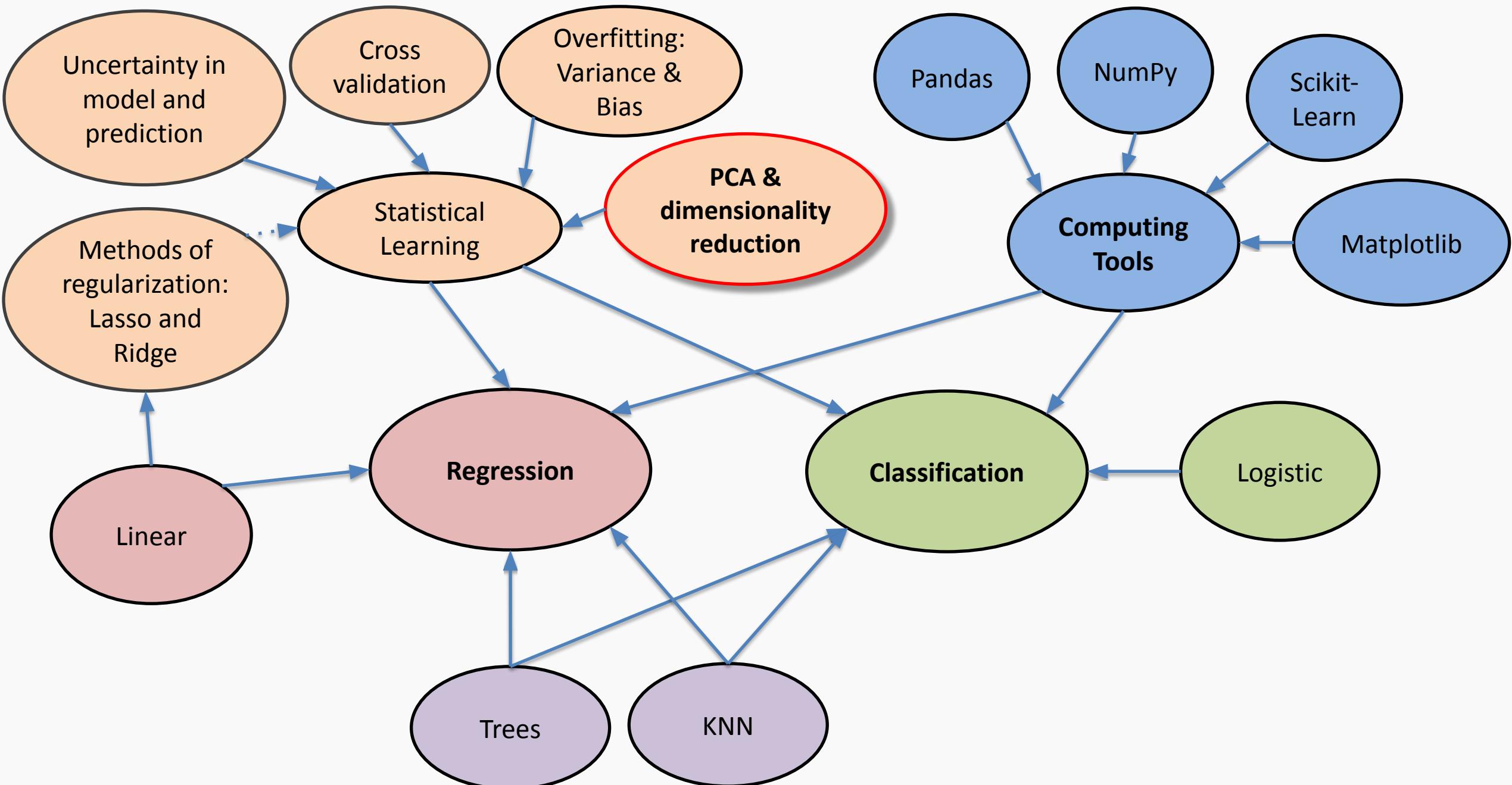
Ridge regularization with a single validation set



Ridge regularization with k-fold cross-validation



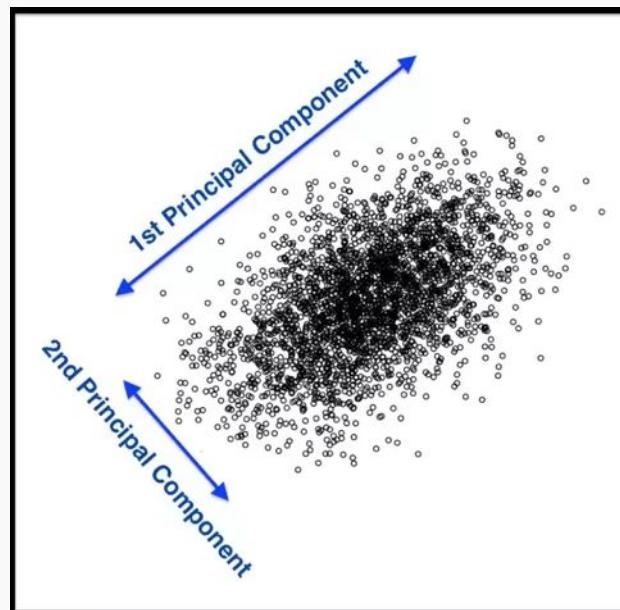




The Intuition Behind PCA

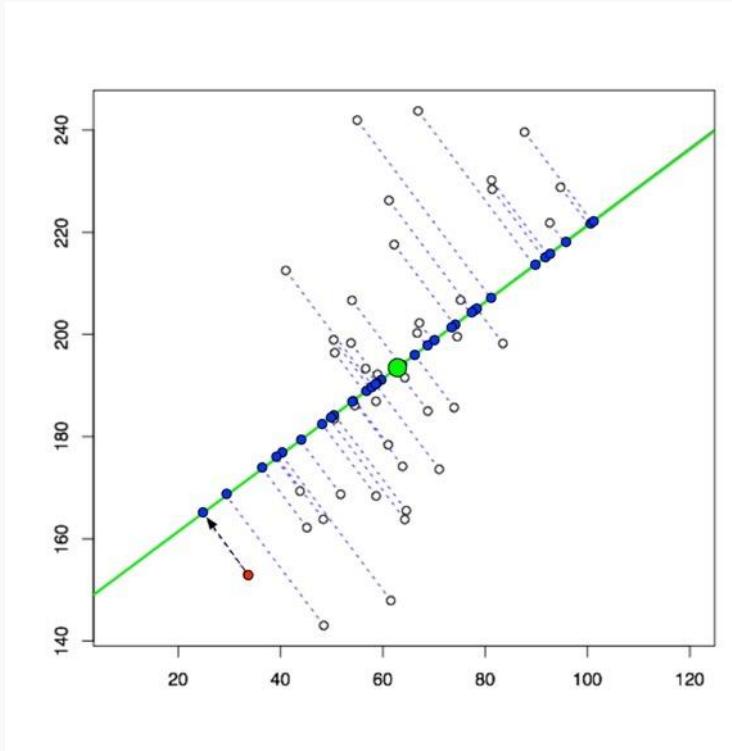
Top PCA components capture the most of amount of variation (interesting features) of the data.

Each component is a linear combination of the original predictors - we visualize them as vectors in the feature space.



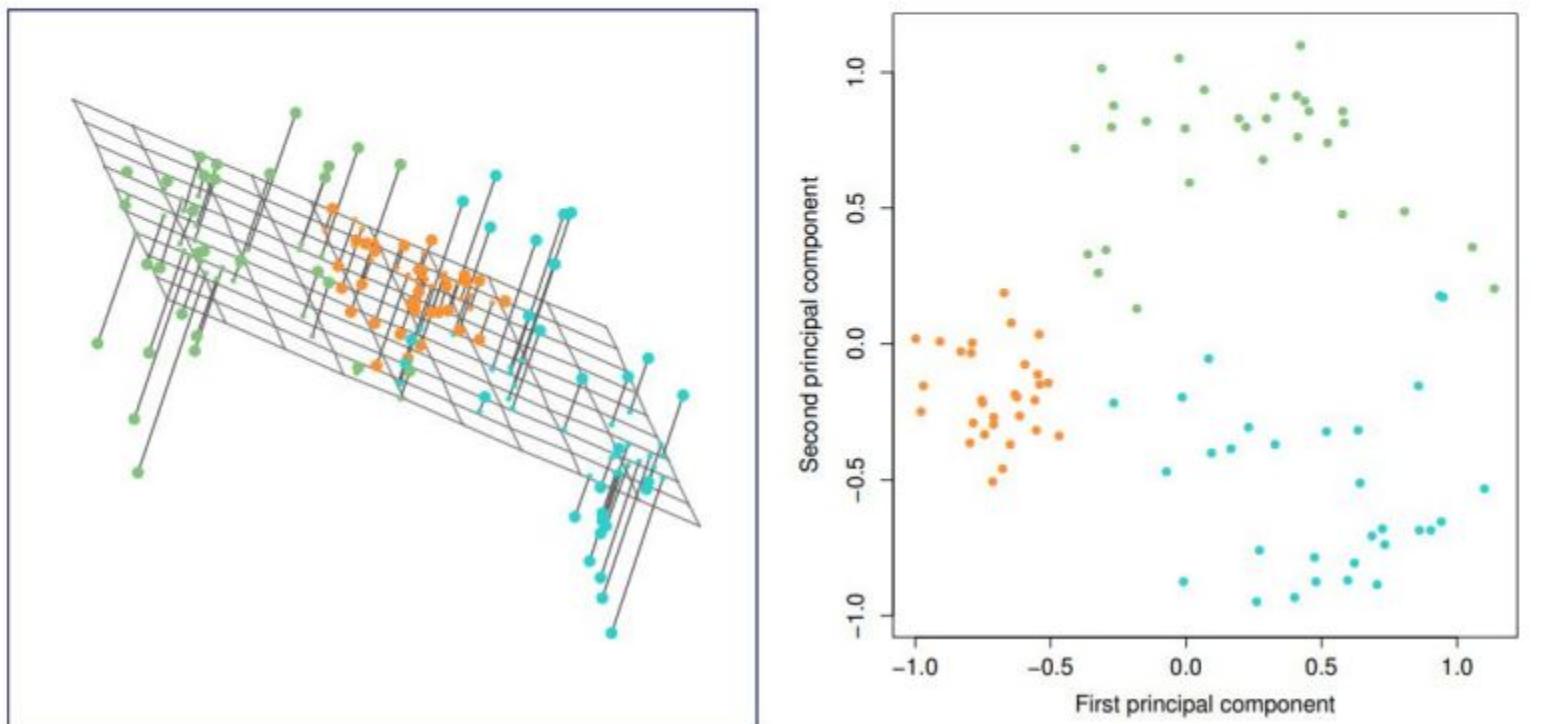
The Intuition Behind PCA (cont.)

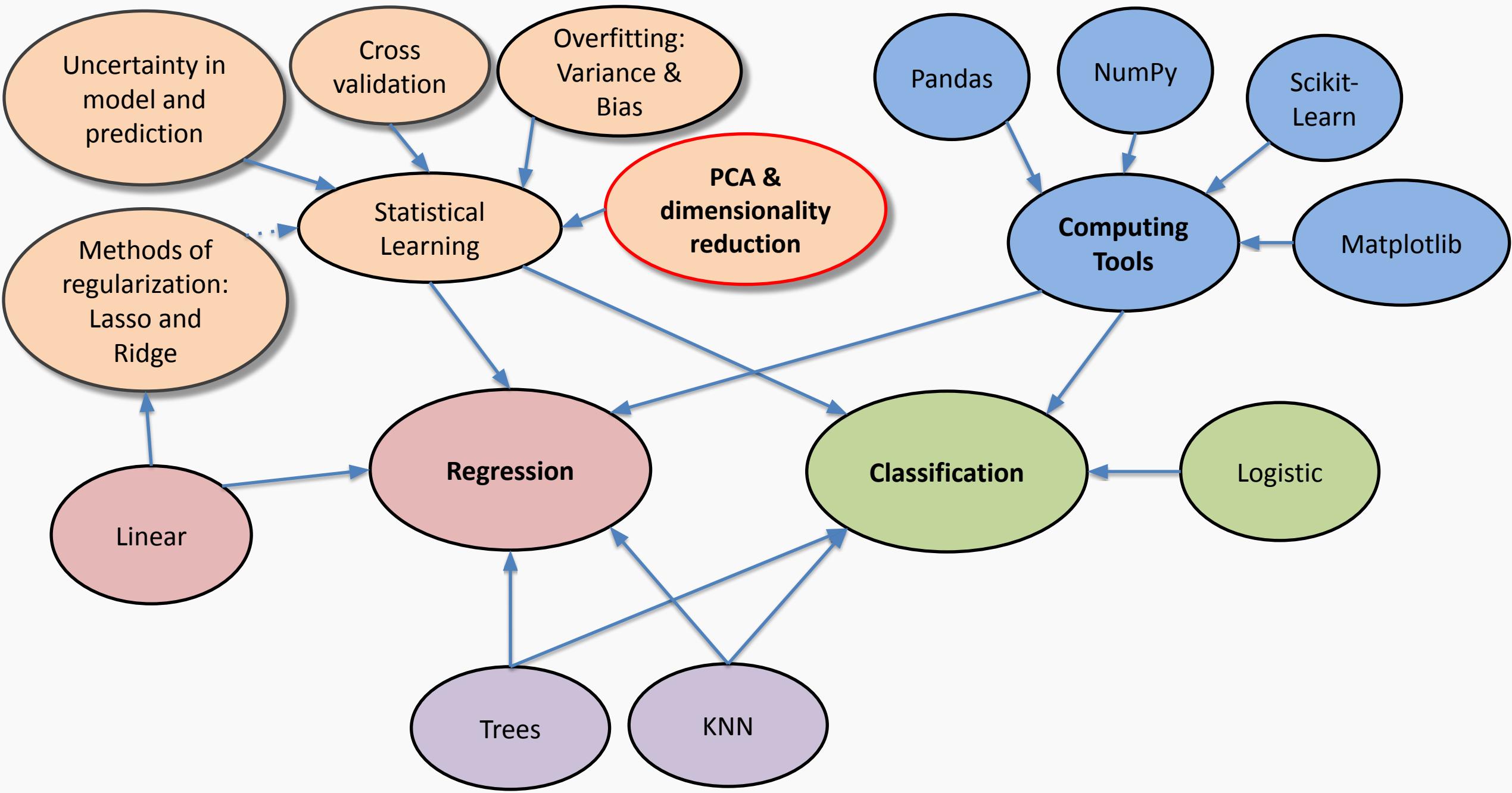
Transforming our observed data means projecting our dataset onto the space defined by the top m PCA components, these components are our new predictors.

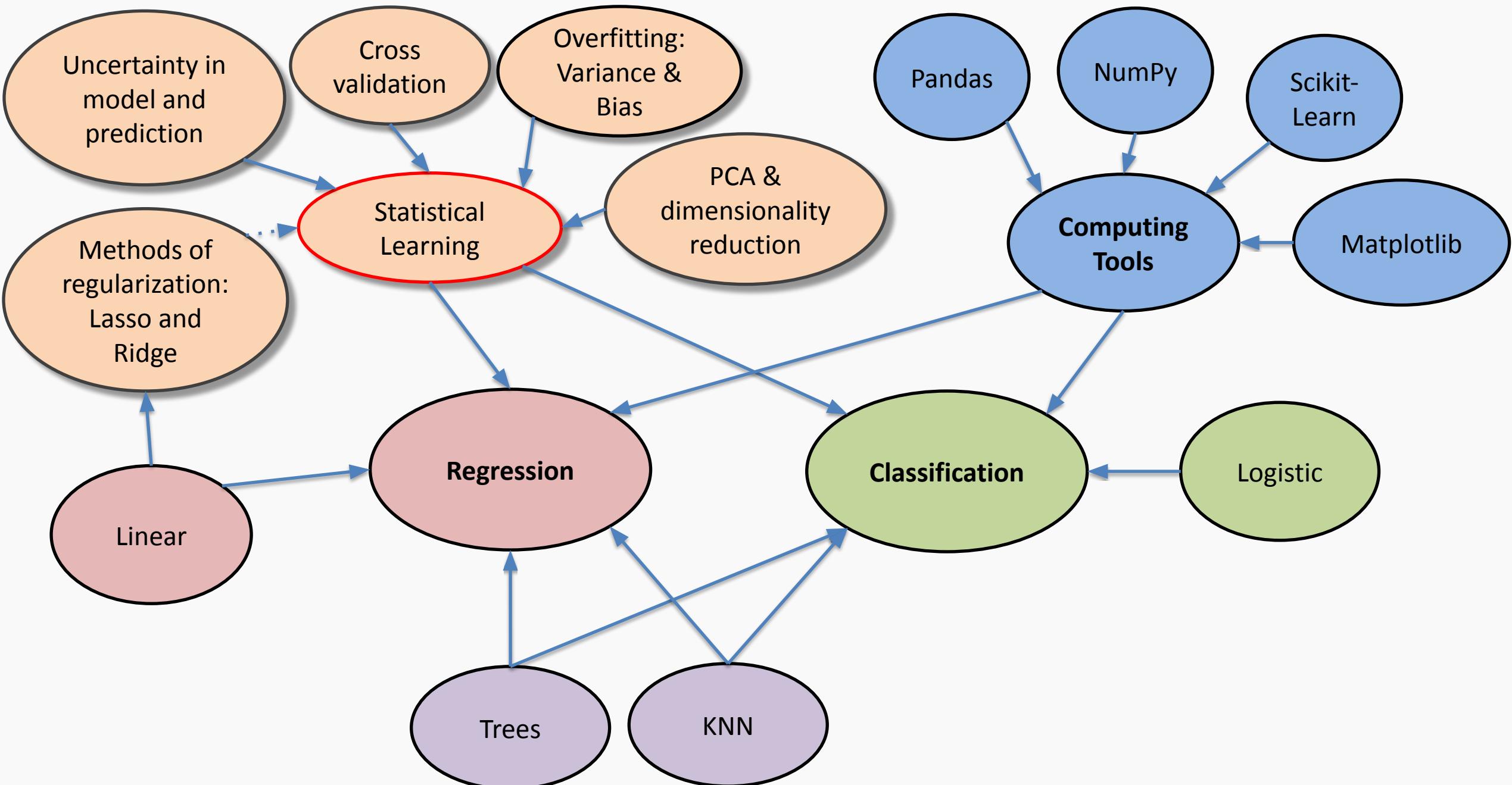


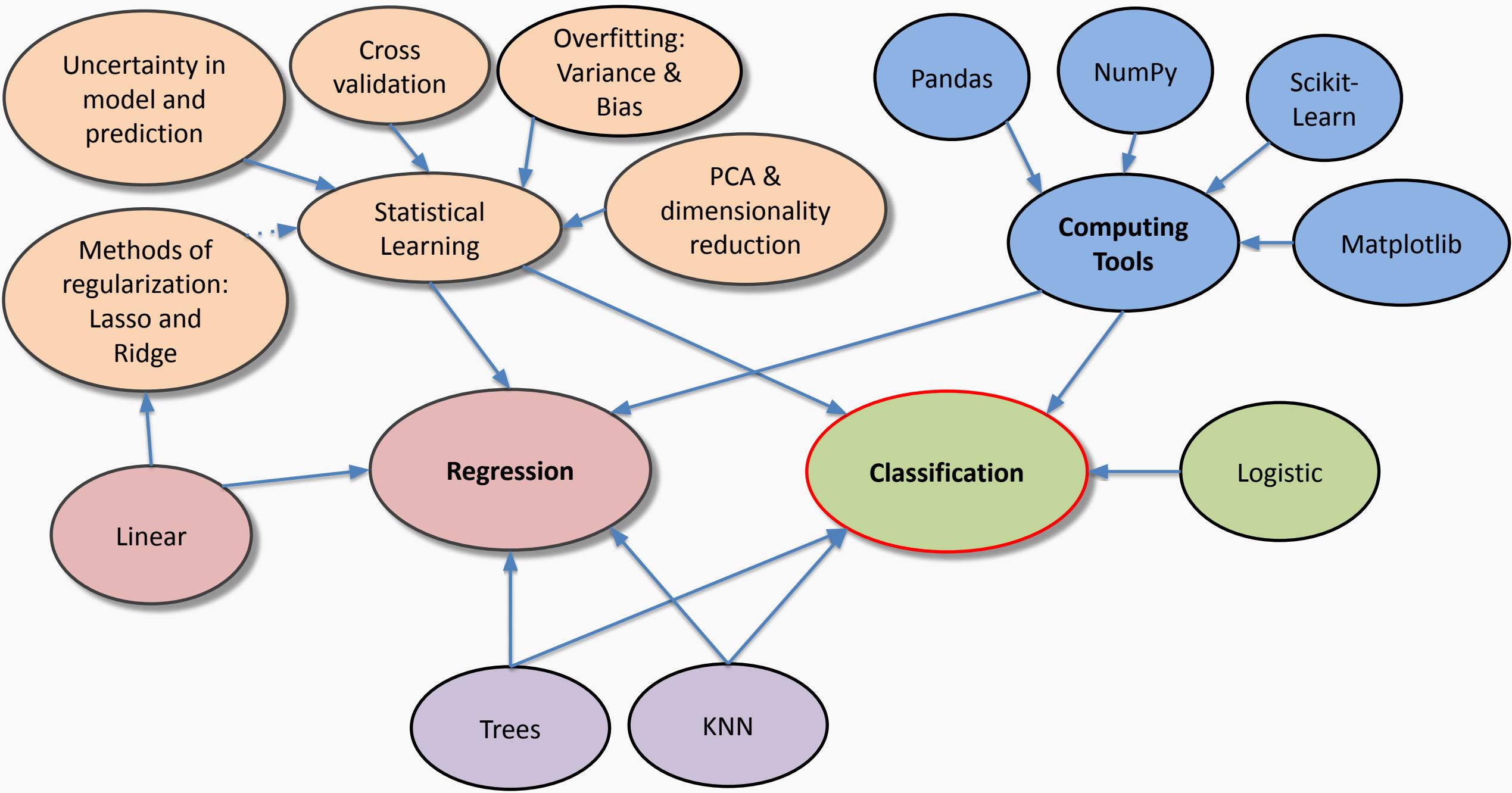
An Alternative Interpretation of PCA

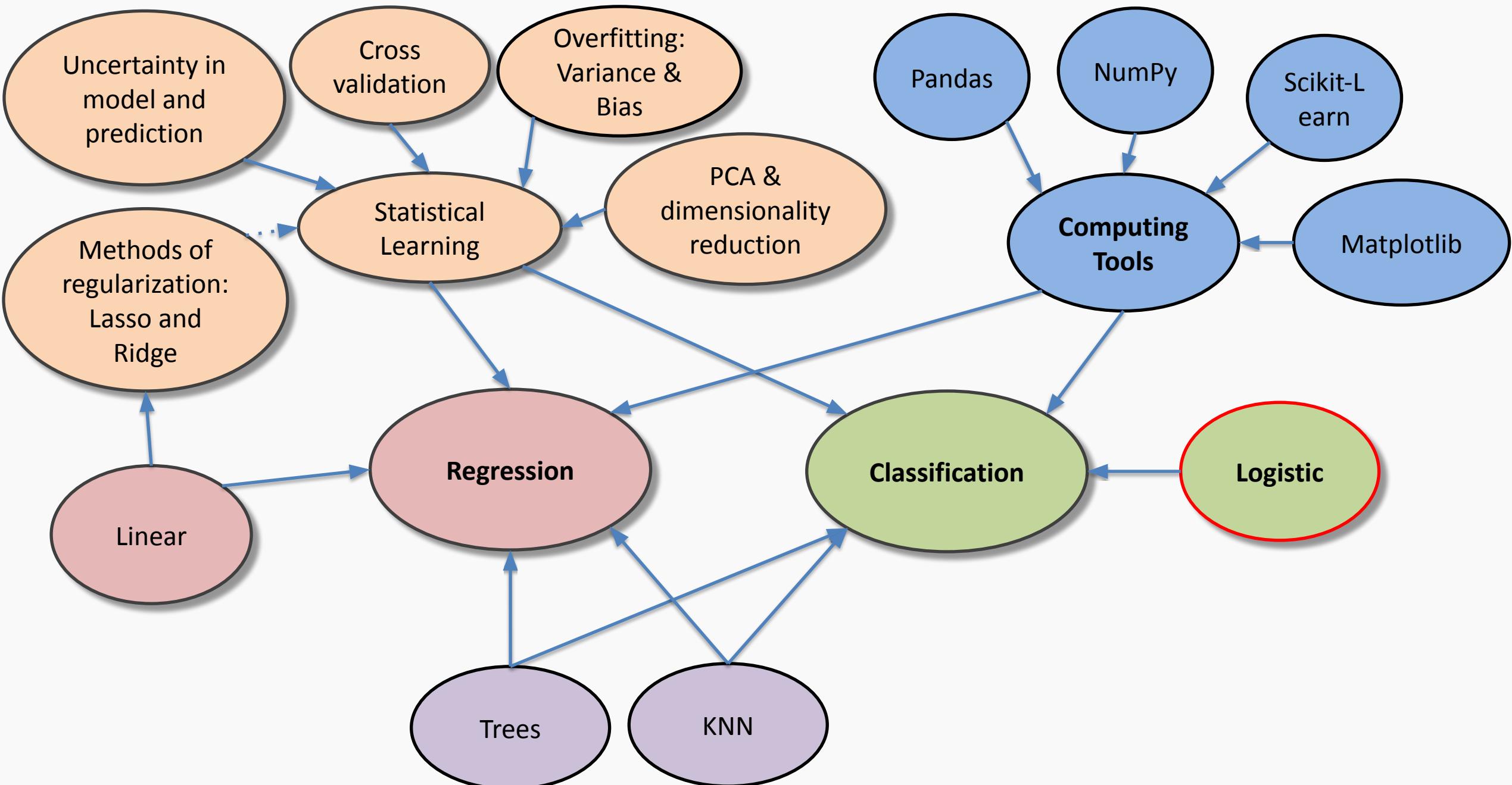
- We've seen an interpretation of PCA as finding the directions in the predictor space along which the data varies the most
- An alternative interpretation is that PCA finds a low-dimensional linear surface which is closest to the data points





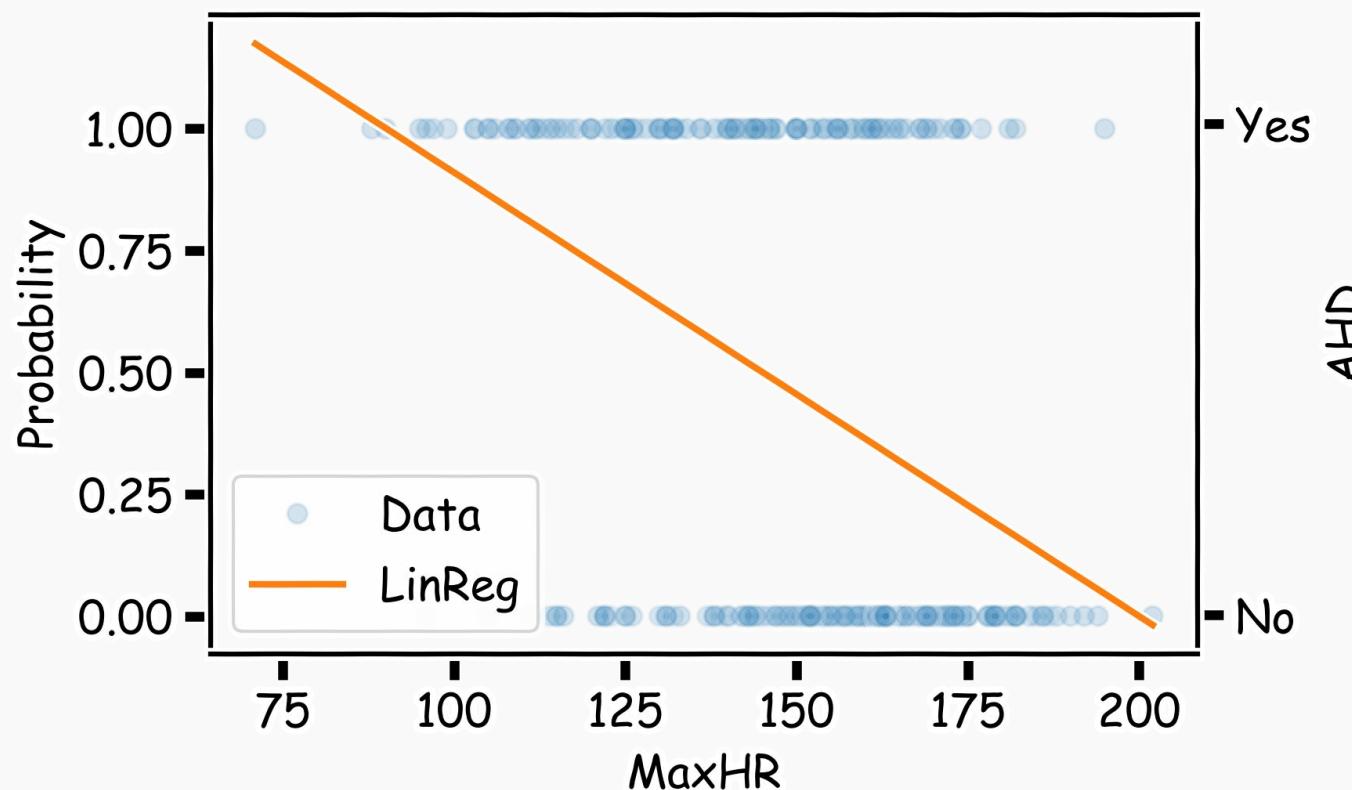






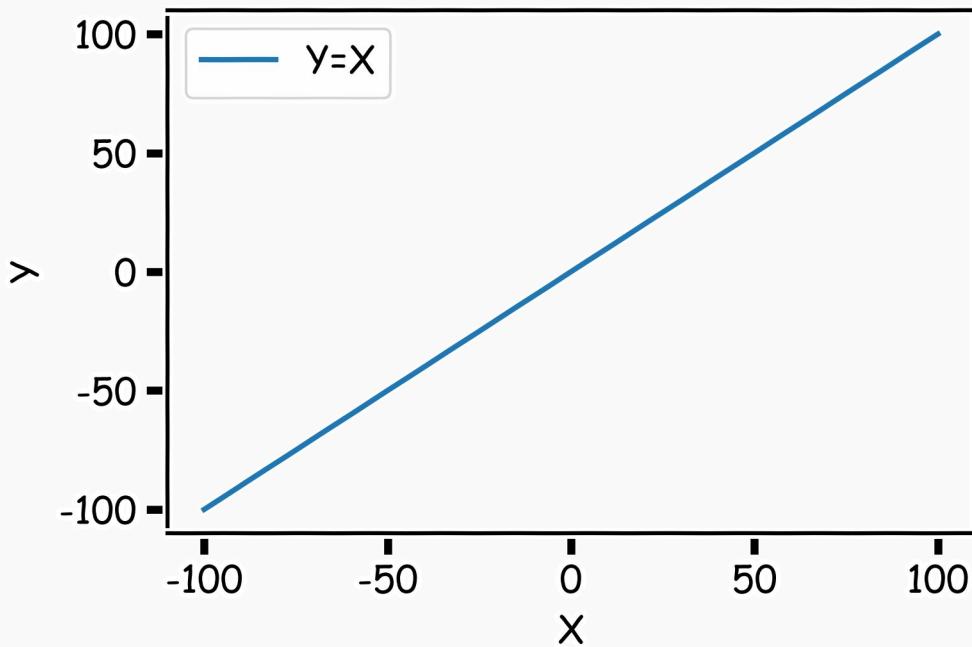
A Simple Classification Problem: Binary Response

What could go wrong with this linear regression model?



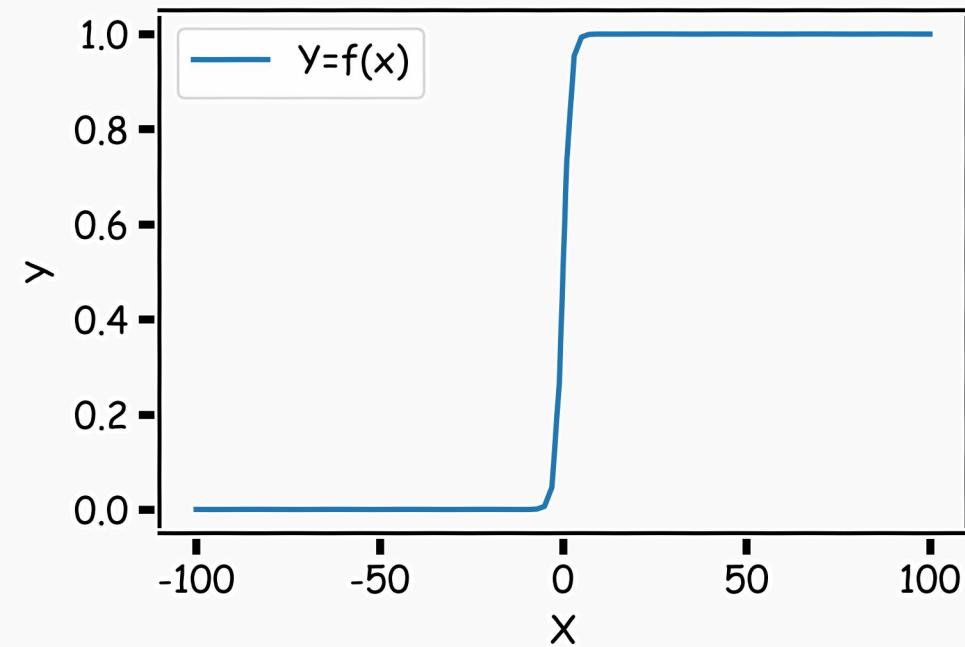
Output Should Be Interpretable As Probabilities

Think of a function that would do this for us

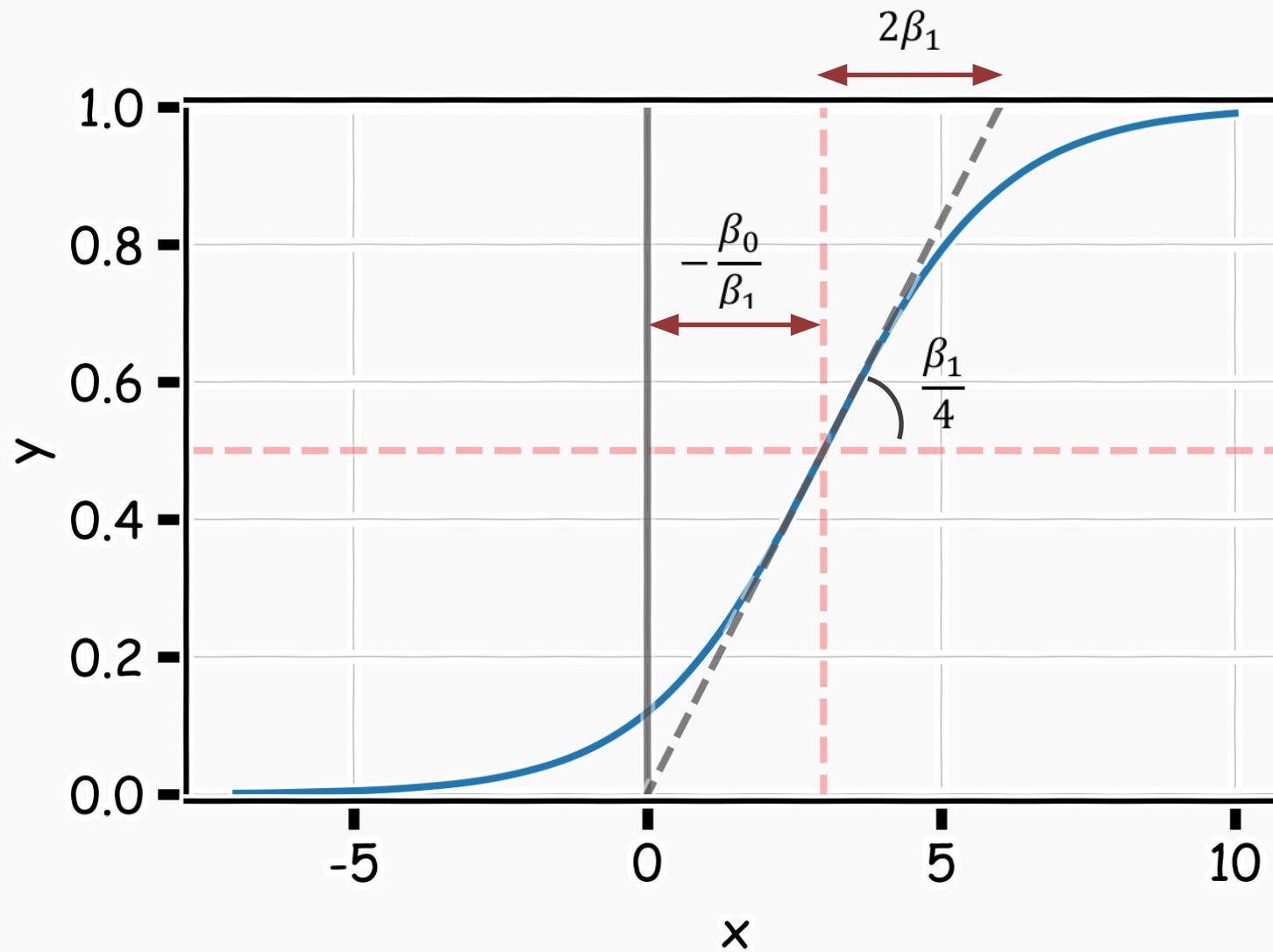


$$Y = f(x)$$

A large grey arrow points from the original linear plot to the right, indicating a transformation.



Logistic Regression



Estimation in Logistic Regression

Probability Mass Function (PMF):

$$P(Y = 1) = p$$

$$P(Y = 0) = 1 - p$$

$$P(Y = y) = p^y(1 - p)^{(1-y)}$$

where:

$$p = P(Y = 1|X = x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

and therefore p depends on X .

Thus not every p_i is the same for each individual measurement.

Likelihood

The likelihood of a single observation for p given x and y is:

$$L(p_i | Y_i) = P(Y_i = y_i) = p_i^{y_i} (1 - p_i)^{1-y_i}$$

Given the observations are independent, what is the likelihood function for p ?

$$L(p | Y) = \prod_i P(Y_i = y_i) = \prod_i p_i^{y_i} (1 - p_i)^{1-y_i}$$

$$l(p | Y) = -\log L(p | Y) = -\sum_i y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

Loss Function

$$l(p|Y) = - \sum_i \left[y_i \log \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_i)}} + (1 - y_i) \log \left(1 - \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_i)}} \right) \right]$$

How do we minimize this?

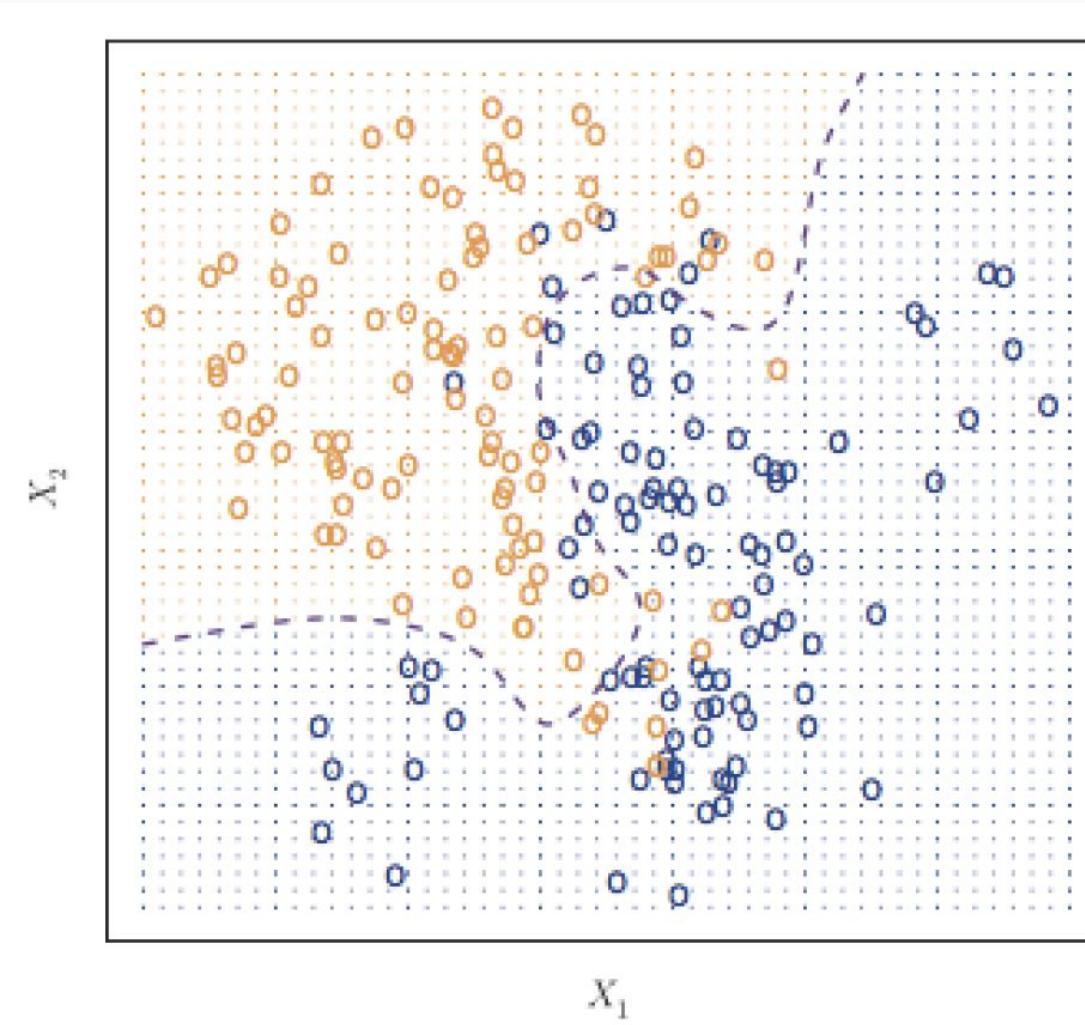
Differentiate, equate to zero and solve for it!

But yikes, does this look messy! It will not necessarily have a closed form solution.

So how do we determine the parameter estimates? Through an iterative approach!

Classifier with two predictors

How can we estimate a classifier, based on logistic regression, for the following plot?



Multiple Logistic Regression

Earlier we saw the general form of simple logistic regression, meaning when there is just one predictor used in the model. What was the model statement (in terms of linear predictors)?

$$\log \left(\frac{P(Y = 1)}{1 - P(Y = 1)} \right) = \beta_0 + \beta_1 X$$

Multiple logistic regression is a generalization to multiple predictors. More specifically we can define a multiple logistic regression model to predict $P(Y = 1)$ as such:

$$\log \left(\frac{P(Y = 1)}{1 - P(Y = 1)} \right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p$$

Regularization in Logistic Regression

A penalty factor can then be added to this loss function and results in a new loss function that penalizes large values of the parameters:

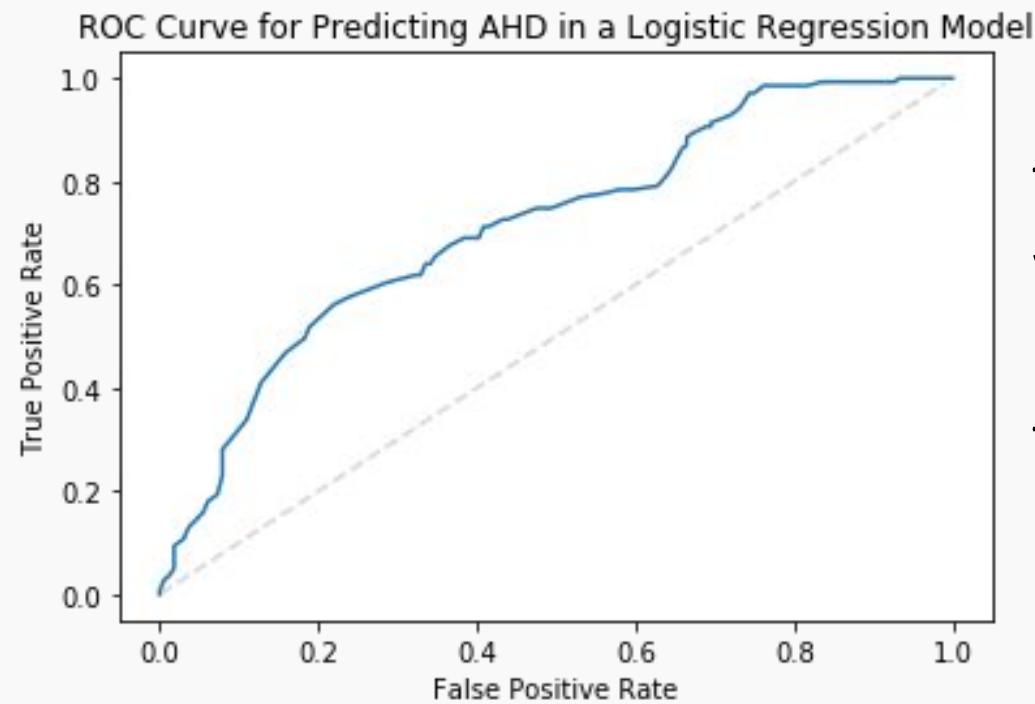
$$\operatorname{argmin}_{\beta_0, \beta_1, \dots, \beta_p} \left[- \sum_{i=1}^n (y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)) + \lambda \sum_{j=1}^p \beta_j^2 \right]$$

The result is just like in linear regression: shrink the parameter estimates towards zero.

In practice, the intercept is usually not part of the penalty factor.

Note: the sklearn package uses a different tuning parameter: instead of λ they use a constant that is essentially $C = \frac{1}{\lambda}$.

ROC Curve Example



The Radio Operator Characteristics (ROC) curve illustrates the trade-off for all possible thresholds chosen for the two types of error (or correct classification).

The vertical axis displays the true positive predictive value and the horizontal axis depicts the true negative predictive value.

The overall performance of a classifier, calculated over all possible thresholds, is given by the **area under the ROC curve (AUC)**.

An ideal ROC curve will hug the top left corner, so the larger the AUC the better the classifier.

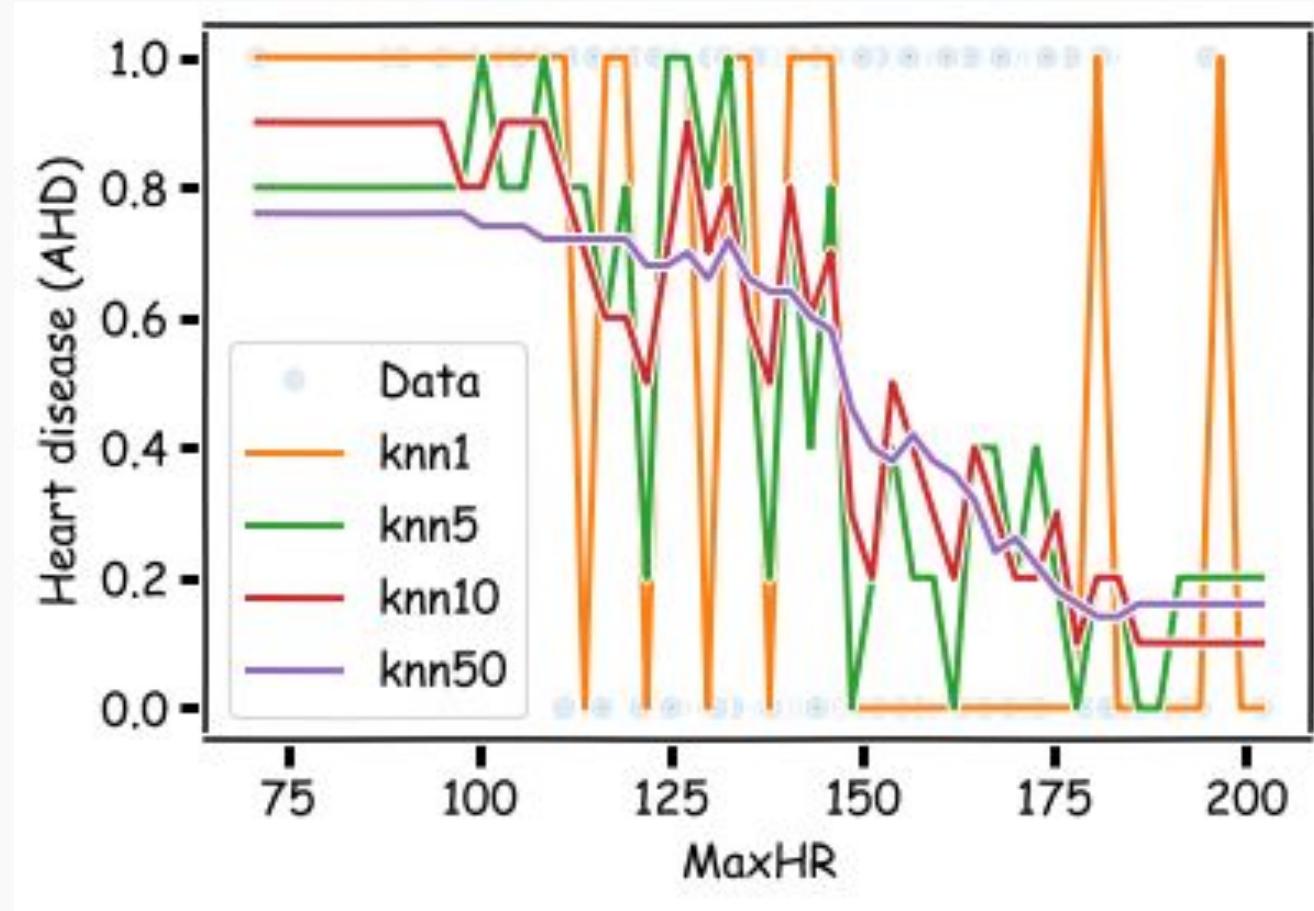
k -NN for Classification: formal definition

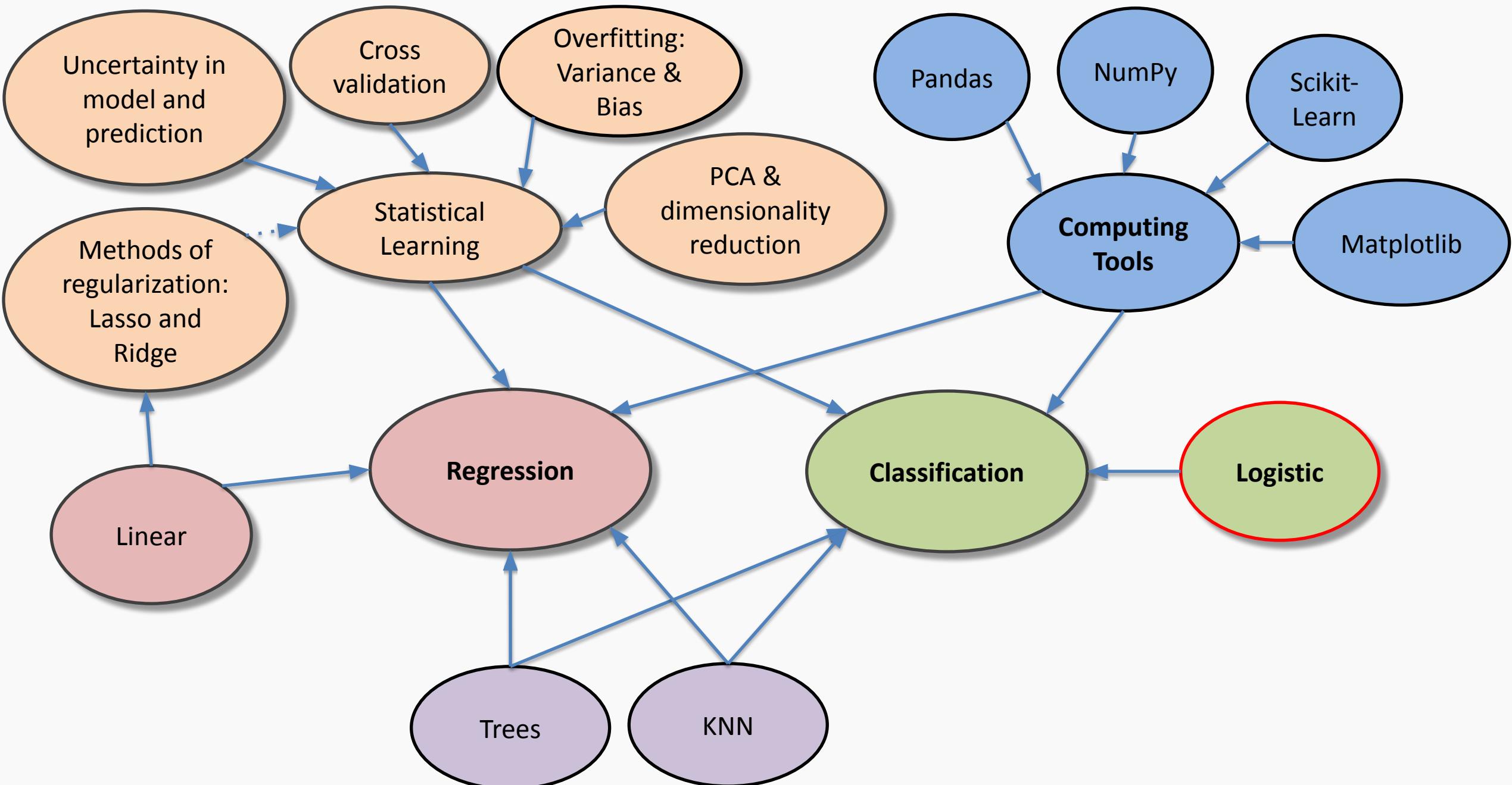
The k -NN classifier first identifies the k points in the training data that are closest to x_0 , represented by \mathcal{N}_0 . It then estimates the conditional probability for class j as the fraction of points in \mathcal{N}_0 whose response values equal j :

$$P(Y = j | X = x_0) = \frac{1}{k} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

Then, the k -NN classifier predicts this new observation, x_0 , to be in the class with largest estimated probability.

Estimated Probabilities in k -NN Classification





Types of Missingness

There are 3 major types of missingness to be concerned about:

1. **Missing Completely at Random (MCAR)** - the probability of missingness in a variable is the same for all units. Like randomly poking holes in a data set.
2. **Missing at Random (MAR)** - the probability of missingness in a variable depends only on available information (in other predictors).
3. **Missing Not at Random (MNAR)** - the probability of missingness depends on information that has not been recorded and this information also predicts the missing values.

What are examples of each these 3 types?

Imputation Methods

There are several different approaches to imputing missing values:

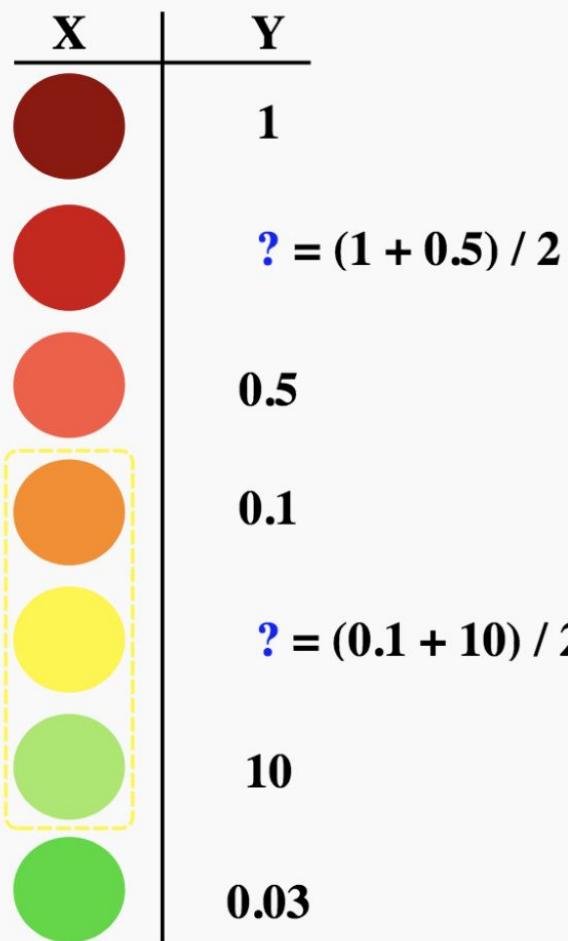
1. **Impute the mean or median** (quantitative) or most common class (categorical) for all missing values in a variable.
2. Create a new variable that is an **indicator of missingness**, and include it in any model to predict the response (also plug in zero or the mean in the actual variable).
3. **Hot deck imputation**: for each missing entry, randomly select an observed entry in the variable and plug it in.
4. **Model the imputation**: plug in predicted values (\hat{y}) from a model based on the other observed predictors.
5. **Model the imputation with uncertainty**: plug in predicted values plus randomness ($\hat{y} + \epsilon$) from a model based on the other observed predictors.

What are the advantages and disadvantages of each approach?

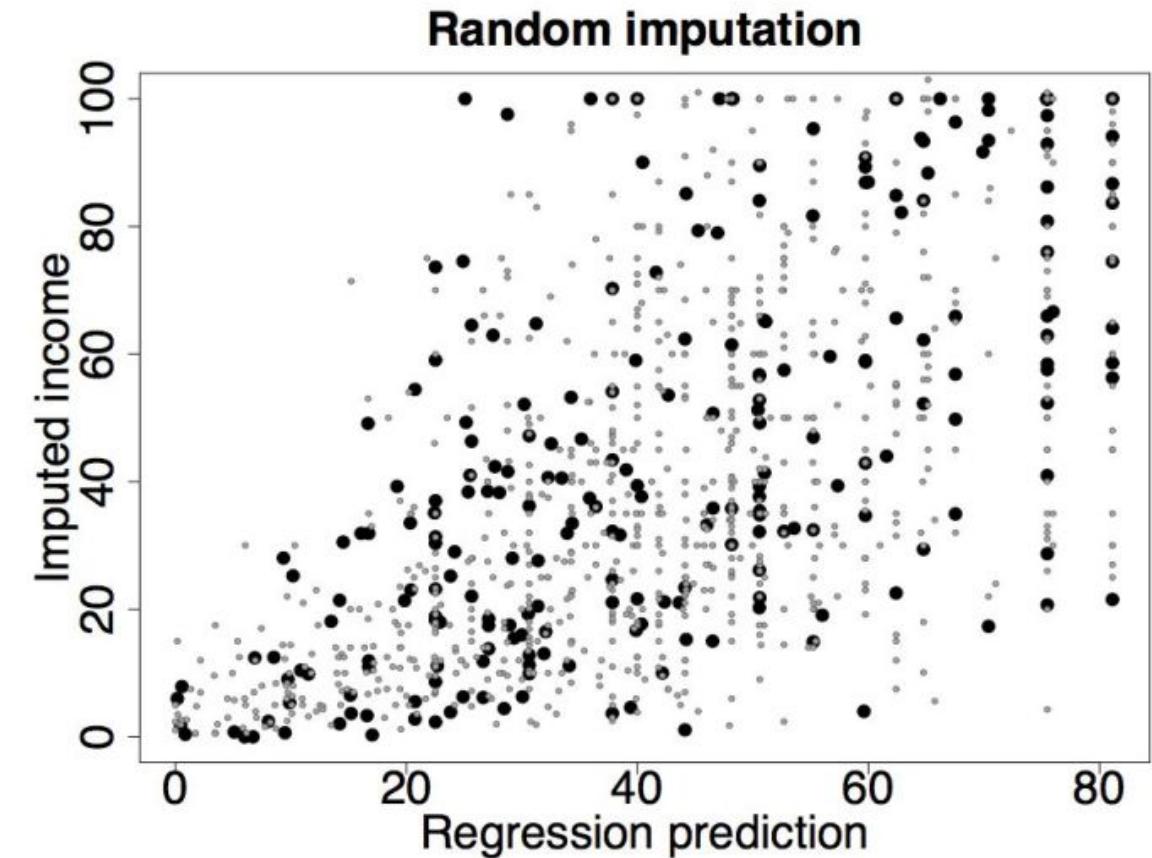
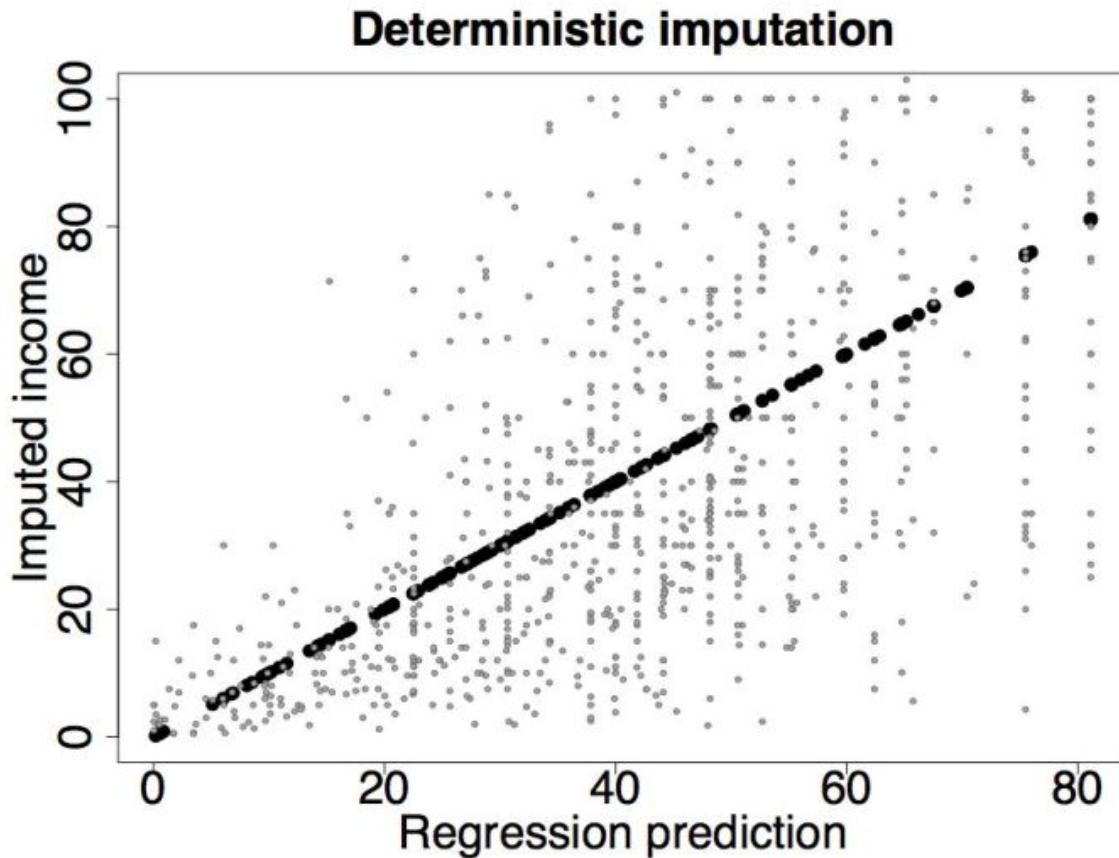


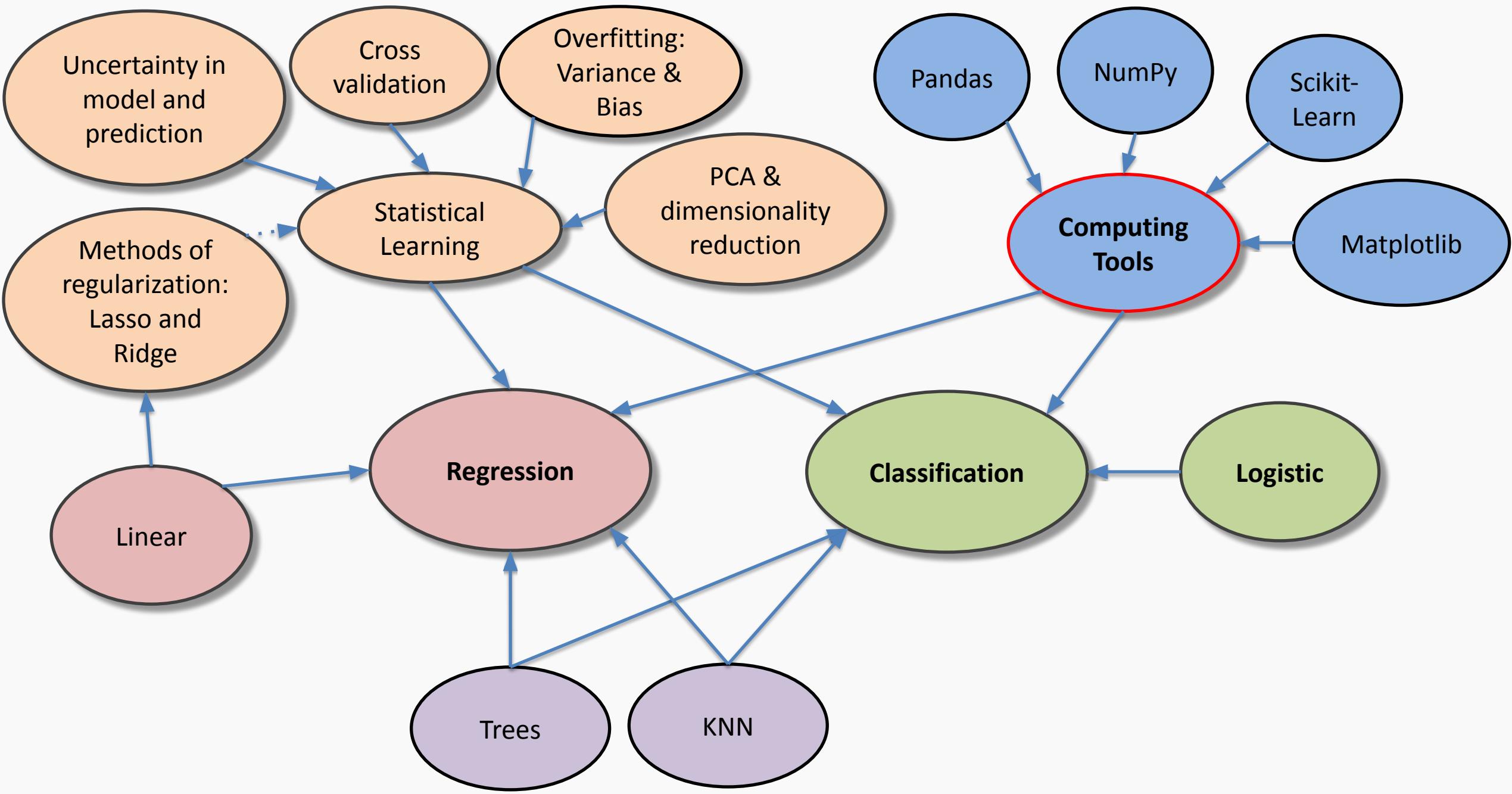
Schematic: imputation through modeling

How do we use models to fill in missing data? Using k -NN for $k = 2$?



Imputation through modeling /w uncertainty: an illustration





Python For Data Science Cheat Sheet

Pandas Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com



Pandas

The Pandas library is built on NumPy and provides easy-to-use data structures and data analysis tools for the Python programming language.



Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

Series

A one-dimensional labeled array capable of holding any data type

a	3
b	-5
c	7
d	4

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame

	Country	Capital	Population
0	Belgium	Brussels	11190846
1	India	New Delhi	1303171035
2	Brazil	Brasilia	207847528

A two-dimensional labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
   >>> 'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
   >>> 'Population': [11190846, 1303171035, 207847528]}
>>> df = pd.DataFrame(data,
   >>> columns=['Country', 'Capital', 'Population'])
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
Read multiple sheets from the same file
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Getting

```
>>> s['b']
-5
>>> df[1:]
   Country    Capital  Population
1  India      New Delhi  1303171035
2  Brazil     Brasilia  207847528
```

Also see NumPy Arrays

Get one element

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0], [0]]
'Belgium'
>>> df.iat[[0], [0]]
'Belgium'
```

By Label

```
>>> df.loc[[0], ['Country']]
'Belgium'
>>> df.at[[0], ['Country']]
'Belgium'
```

By Label/Position

```
>>> df.ix[2]
   Country    Brazil
   Capital    Brasilia
   Population 207847528
```

```
>>> df.ix[:, 'Capital']
0    Brussels
1   New Delhi
2    Brasilia
```

```
>>> df.ix[1, 'Capital']
'New Delhi'
```

Boolean Indexing

```
>>> s[~(s > 1)]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population']>1200000000]
```

Setting

```
>>> s['a'] = 6
```

Select single value by row & column

Select single value by row & column labels

Select single row of subset of rows

Select a single column of subset of columns

Select rows and columns

Series s where value is not >1
s where value is <-1 or >2
Use filter to adjust DataFrame

Set index a of Series s to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```

Drop values from rows (axis=0)
Drop values from columns(axis=1)

Sort & Rank

```
>>> df.sort_index()
>>> df.sort_values(by='Country')
>>> df.rank()
```

Sort by labels along an axis
Sort by the values along an axis
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows,columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min() / df.max()
>>> df.idxmin() / df.idxmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values
Cummulative sum of values
Minimum/maximum values
Minimum/Maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
a    10.0
b    NaN
c     5.0
d     7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
a    10.0
b    -5.0
c     5.0
d     7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```



Python For Data Science Cheat Sheet

NumPy Basics

Learn Python for Data Science [Interactively](#) at www.DataCamp.com



NumPy

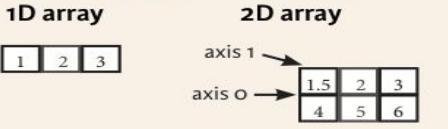
The NumPy library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```



NumPy Arrays



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]),
      dtype = float)
```

Initial Placeholders

```
>>> np.zeros((3,4))          Create an array of zeros
>>> np.ones((2,3),dtype=np.int16) Create an array of ones
>>> d = np.arange(10,25,5)    Create an array of evenly
                             spaced values (step value)
>>> np.linspace(0,2,9)       Create an array of evenly
                             spaced values (number of samples)
>>> e = np.full((2,2),7)     Create a constant array
>>> f = np.eye(2)            Create a 2X2 identity matrix
>>> np.random.random((2,2))  Create an array with random values
>>> np.empty((3,2))          Create an empty array
```

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

```
>>> np.int64
        Signed 64-bit integer types
>>> np.float32
        Standard double-precision floating point
>>> np.complex
        Complex numbers represented by 128 floats
>>> np.bool_
        Boolean type storing TRUE and FALSE values
>>> np.object_
        Python object type
>>> np.string_
        Fixed-length string type
>>> np.unicode_
        Fixed-length unicode type
```

Inspecting Your Array

```
>>> a.shape
        Array dimensions
>>> len(a)
        Length of array
>>> b.ndim
        Number of array dimensions
>>> e.size
        Number of array elements
>>> b.dtype
        Data type of array elements
>>> b.dtype.name
        Name of data type
>>> b.astype(int)
        Convert an array to a different type
```

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b
        array([[-0.5,  0. ,  0. ],
               [-3. , -3. , -3. ]])
>>> np.subtract(a,b)
>>> b + a
        array([[ 2.5,  4. ,  6. ],
               [ 5. ,  7. ,  9. ]])
>>> np.add(b,a)
>>> a / b
        array([[ 0.66666667,  1. ,
               [ 0.25,  0.4,  1. ,
               [ 0.5,  1. ]])
>>> np.divide(a,b)
>>> a * b
        array([[ 1.5,  4. ,  9. ],
               [ 4. , 10. , 18. ]])
>>> np.multiply(a,b)
>>> np.exp(b)
>>> np.sqrt(b)
>>> np.sin(a)
>>> np.cos(b)
>>> np.log(a)
>>> e.dot(f)
        array([[ 7.,  7.],
               [ 7.,  7.]])
```

Subtraction

Addition

Division

Multiplication

Multiplication
Exponentiation
Square root
Print sines of an array
Element-wise cosine
Element-wise natural logarithm
Dot product

Comparison

```
>>> a == b
        array([[False,  True,  True],
               [False, False, False]], dtype=bool)
>>> a < 2
        array([True, False, False], dtype=bool)
>>> np.array_equal(a, b)
```

Element-wise comparison

Element-wise comparison

Array-wise comparison

Aggregate Functions

```
>>> a.sum()
        Array-wise sum
>>> a.min()
        Array-wise minimum value
>>> b.max(axis=0)
        Maximum value of an array row
>>> b.cumsum(axis=1)
        Cumulative sum of the elements
>>> a.mean()
        Mean
>>> b.median()
        Median
>>> a.corrcoef()
        Correlation coefficient
>>> np.std(b)
        Standard deviation
```

Copying Arrays

```
>>> h = a.view()
        Create a view of the array with the same data
>>> np.copy(a)
        Create a copy of the array
>>> h = a.copy()
        Create a deep copy of the array
```

Sorting Arrays

```
>>> a.sort()
        Sort an array
>>> c.sort(axis=0)
        Sort the elements of an array's axis
```

Subsetting, Slicing, Indexing

Subsetting

```
>>> a[2]
        3
>>> b[1,2]
        6.0
```

1	2	3
1.5	2	3
4	5	6

Select the element at the 2nd index
Select the element at row 0 column 2 (equivalent to `b[1][2]`)

Slicing

```
>>> a[0:2]
        array([1, 2])
>>> b[0:2,1]
        array([ 2.,  5.])
>>> b[:,1]
        array([[ 1.5,  2.,  3.]])
>>> c[1,:,:]
        array([[ 3.,  2.,  1.],
               [ 4.,  5.,  6.]])
>>> a[ : :-1]
        array([3, 2, 1])
```

1	2	3
1.5	2	3
4	5	6

Select items at index 0 and 1
Select items at rows 0 and 1 in column 1
Select all items at row 0 (equivalent to `b[0:1, :]`)
Same as `[1, :, :]`

Reversed array

```
>>> a[ : :-1]
```

Reversed array `a`

Boolean Indexing

```
>>> a[a<2]
        array([1])
```

1	2	3
1.5	2	1
4	5	6

Select elements from `a` less than 2
Select elements `(1,0), (0,1), (1,2)` and `(0,0)`
Select a subset of the matrix's rows and columns

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b)
>>> i.T
```

Permute array dimensions
Permute array dimensions

Changing Array Shape

```
>>> b.ravel()
>>> g.reshape(3,-2)
```

Flatten the array
Reshape, but don't change data

Adding/Removing Elements

```
>>> h.resize((2,6))
>>> np.append(h,g)
>>> np.insert(a, 1, 5)
>>> np.delete(a, [1])
```

Return a new array with shape `(2,6)`
Append items to an array
Insert items in an array
Delete items from an array

Combining Arrays

```
>>> np.concatenate((a,d),axis=0)
        array([ 1,  2,  3, 10, 15, 20])
>>> np.vstack((a,b))
        array([[ 1.,  2.,  3.],
               [ 1.5,  2.,  3.],
               [ 4.,  5.,  6.]])
>>> np.r_[e,f]
>>> np.hstack((e,f))
        array([[ 7.,  7.,  1.,  0.],
               [ 7.,  7.,  0.,  1.]])
>>> np.column_stack((a,d))
        array([[ 1,  10],
               [ 2,  15],
               [ 3,  20]])
>>> np.c_[a,d]
```

Concatenate arrays
Stack arrays vertically (row-wise)

Splitting Arrays

```
>>> np.hsplit(a,3)
        [array([1]), array([2]), array([3])]
>>> np.vsplit(c,2)
        [array([[ 1.5,  2.,  1. ],
               [ 4.,  5.,  6.]]),
         array([[ 3.,  2.,  1. ],
               [ 4.,  5.,  6.]])]
```

Split the array horizontally at the 3rd index
Split the array vertically at the 2nd index



Python For Data Science Cheat Sheet

SciPy - Linear Algebra

Learn More Python for Data Science [Interactively](#) at www.datacamp.com



SciPy

The SciPy library is one of the core packages for scientific computing that provides mathematical algorithms and convenience functions built on the NumPy extension of Python.



Interacting With NumPy

Also see NumPy

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> b = np.array([(1+5j,2j,3j), (4j,5j,6j)])
>>> c = np.array([(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)])
```

Index Tricks

>>> np.mgrid[0:5,0:5]	Create a dense meshgrid
>>> np.ogrid[0:2,0:2]	Create an open meshgrid
>>> np.r_[3,[0]*5,-1:1:10j]	Stack arrays vertically (row-wise)
>>> np.c_[b,c]	Create stacked column-wise arrays

Shape Manipulation

>>> np.transpose(b)	Permute array dimensions
>>> b.flatten()	Flatten the array
>>> np.hstack((b,c))	Stack arrays horizontally (column-wise)
>>> np.vstack((a,b))	Stack arrays vertically (row-wise)
>>> np.hsplit(c,2)	Split the array horizontally at the 2nd index
>>> np.vsplit(d,2)	Split the array vertically at the 2nd index

Polynomials

```
>>> from numpy import poly1d
>>> p = poly1d([3,4,5])
```

Create a polynomial object

Vectorizing Functions

```
>>> def myfunc(a):
...     if a < 0:
...         return a**2
...     else:
...         return a/2
>>> np.vectorize(myfunc)
```

Vectorize functions

Type Handling

>>> np.real(b)	Return the real part of the array elements
>>> np.imag(b)	Return the imaginary part of the array elements
>>> np.real_if_close(c,tol=1000)	Return a real array if complex parts close to 0
>>> np.cast['f'](np.pi)	Cast object to a data type

Other Useful Functions

>>> np.angle(b,deg=True)	Return the angle of the complex argument
>>> g = np.linspace(0,np.pi,num=5)	Create an array of evenly spaced values (number of samples)
>>> g[3:] += np.pi	Unwrap
>>> np.unwrap(g)	Create an array of evenly spaced values (log scale)
>>> np.logspace(0,10,3)	Return values from a list of arrays depending on conditions
>>> np.select([c<4],[c*2])	Factorial
>>> misc.factorial(a)	Combine N things taken at k time
>>> misc.comb(10,3,exact=True)	Weights for N-point central derivative
>>> misc.central_diff_weights(3)	Find the n-th derivative of a function at a point
>>> misc.derivative(myfunc,1.0)	

Linear Algebra

You'll use the `linalg` and `sparse` modules. Note that `scipy.linalg` contains and expands on `numpy.linalg`.

Also see NumPy

```
>>> from scipy import linalg, sparse
```

Creating Matrices

```
>>> A = np.matrix(np.random.random((2,2)))
>>> B = np.asmatrix(b)
>>> C = np.mat(np.random.random((10,5)))
>>> D = np.mat([[3,4], [5,6]])
```

Basic Matrix Routines

Inverse

```
>>> A.I
>>> linalg.inv(A)
```

Transposition

```
>>> A.T
>>> A.H
```

Trace

```
>>> np.trace(A)
```

Norm

```
>>> linalg.norm(A)
>>> linalg.norm(A,1)
>>> linalg.norm(A,np.inf)
```

Rank

```
>>> np.linalg.matrix_rank(C)
```

Determinant

```
>>> linalg.det(A)
```

Solving linear problems

```
>>> linalg.solve(A,b)
>>> E = np.mat(a).T
>>> linalg.lstsq(F,E)
```

Generalized inverse

```
>>> linalg.pinv(C)
>>> linalg.pinv2(C)
```

Creating Sparse Matrices

```
>>> F = np.eye(3, k=1)
>>> G = np.mat(np.identity(2))
>>> C[C > 0.5] = 0
>>> H = sparse.csr_matrix(C)
>>> I = sparse.csc_matrix(D)
>>> J = sparse.dok_matrix(A)
>>> E.todense()
>>> sparse.isspmatrix_csc(A)
```

Sparse Matrix Routines

Inverse

```
>>> sparse.linalg.inv(I)
```

Norm

```
>>> sparse.linalg.norm(I)
```

Solving linear problems

```
>>> sparse.linalg.spsolve(H,I)
```

Sparse Matrix Functions

```
>>> sparse.linalg.expm(I)      Sparse matrix exponential
```

Asking For Help

```
>>> help(scipy.linalg.diagsvd)
```

```
>>> np.info(np.matrix)
```

Matrix Functions

Addition

```
>>> np.add(A,D)
```

Subtraction

```
>>> np.subtract(A,D)
```

Division

```
>>> np.divide(A,D)
```

Multiplication

```
>>> A @ D
```

```
>>> np.multiply(D,A)
```

```
>>> np.dot(A,D)
```

```
>>> np.vdot(A,D)
```

```
>>> np.inner(A,D)
```

```
>>> np.outer(A,D)
```

```
>>> np.tensordot(A,D)
```

```
>>> np.kron(A,D)
```

Exponential Functions

```
>>> linalg.expm(A)
```

```
>>> linalg.expm2(A)
```

```
>>> linalg.expm3(D)
```

Logarithm Function

```
>>> linalg.logm(A)
```

Trigonometric Functions

```
>>> linalg.sinm(D)
```

```
>>> linalg.cosm(D)
```

```
>>> linalg.tanm(A)
```

Hyperbolic Trigonometric Functions

```
>>> linalg.sinhm(D)
```

```
>>> linalg.coshm(D)
```

```
>>> linalg.tanhm(A)
```

Matrix Sign Function

```
>>> np.signm(A)
```

Matrix Square Root

```
>>> linalg.sqrtm(A)
```

Arbitrary Functions

```
>>> linalg.funm(A, lambda x: x**x)
```

Decompositions

Eigenvalues and Eigenvectors

```
>>> la, v = linalg.eig(A)
```

```
>>> l1, l2 = la
```

```
>>> v[:,0]
```

```
>>> v[:,1]
```

```
>>> linalg.eigvals(A)
```

Singular Value Decomposition

```
>>> U,s,Vh = linalg.svd(B)
```

```
>>> M,N = B.shape
```

```
>>> Sig = linalg.diagsvd(s,M,N)
```

LU Decomposition

```
>>> P,L,U = linalg.lu(C)
```

Sparse Matrix Decompositions

```
>>> la, v = sparse.linalg.eigs(F,1)
```

```
>>> sparse.linalg.svds(H, 2)
```

Solve ordinary or generalized eigenvalue problem for square matrix

Unpack eigenvalues

First eigenvector

Second eigenvector

Unpack eigenvalues

Singular Value Decomposition (SVD)

Construct sigma matrix in SVD

LU Decomposition

Eigenvalues and eigenvectors SVD

DataCamp

Learn Python for Data Science [Interactively](#)

Python For Data Science Cheat Sheet

Matplotlib

Learn Python **Interactively** at www.DataCamp.com



Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.



1 Prepare The Data

Also see [Lists & NumPy](#)

1D Data

```
>>> import numpy as np  
>>> x = np.linspace(0, 10, 100)  
>>> y = np.cos(x)  
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))  
>>> data2 = 3 * np.random.random((10, 10))  
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]  
>>> U = -1 - X**2 + Y  
>>> V = 1 + X - Y**2  
>>> from matplotlib.cbook import get_sample_data  
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2 Create Plot

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure()  
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

Axes

All plotting is done with respect to an Axes. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()  
>>> ax1 = fig.add_subplot(221) # row-col-num  
>>> ax3 = fig.add_subplot(212)  
>>> fig3, axes = plt.subplots(nrows=2, ncols=2)  
>>> fig4, axes2 = plt.subplots(ncols=3)
```

3 Plotting Routines

1D Data

```
>>> fig, ax = plt.subplots()  
>>> lines = ax.plot(x,y)  
>>> ax.scatter(x,y)  
>>> axes[0,0].bar([1,2,3],[3,4,5])  
>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])  
>>> axes[1,1].axhline(0.45)  
>>> axes[0,1].axvline(0.65)  
>>> ax.fill(x,y,color='blue')  
>>> ax.fill_between(x,y,color='yellow')
```

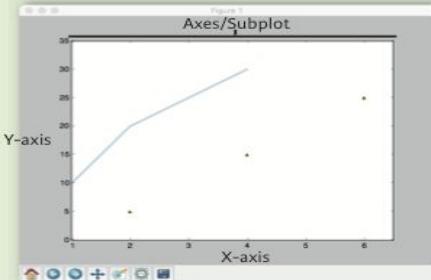
2D Data or Images

```
>>> fig, ax = plt.subplots()  
>>> im = ax.imshow(img,  
                  cmap='gist_earth',  
                  interpolation='nearest',  
                  vmin=-2,  
                  vmax=2)
```

Colormapped or RGB arrays

Plot Anatomy & Workflow

Plot Anatomy



Figure

Workflow

The basic steps to creating plots with matplotlib are:

- 1 Prepare data
 - 2 Create plot
 - 3 Plot
 - 4 Customize plot
 - 5 Save plot
 - 6 Show plot
- ```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4] Step 1
>>> y = [10,20,25,30]
>>> fig = plt.figure() Step 2
>>> ax = fig.add_subplot(111) Step 3
>>> ax.plot(x, y, color='lightblue', linewidth=3) Step 3, 4
>>> ax.scatter([2,4,6],
 [5,15,25],
 color='darkgreen',
 marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show() Step 6
```

## 4 Customize Plot

### Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
 cmap='seismic')
```

### Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

### Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,Y,'--',x*x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

### Text & Annotations

```
>>> ax.text(1,-2.1,
 'Example Graph',
 style='italic')
>>> ax.annotate("Sine",
 xy=(8, 0),
 xycoords='data',
 xytext=(10.5, 0),
 textcoords='data',
 arrowprops=dict(arrowstyle="->",
 connectionstyle="arc3"),
)
```

### Vector Fields

```
>>> axes[0,1].arrow(0,0,0.5,0.5)
>>> axes[1,1].quiver(y,z)
>>> axes[0,1].streamplot(X,Y,U,V)
```

Add an arrow to the axes  
Plot a 2D field of arrows  
Plot a 2D field of arrows

### Data Distributions

```
>>> ax1.hist(y)
>>> ax3.boxplot(y)
>>> ax3.violinplot(z)
```

Plot a histogram  
Make a box and whisker plot  
Make a violin plot

Pseudocolor plot of 2D array  
Pseudocolor plot of 2D array  
Plot contours  
Plot filled contours  
Label a contour plot

## 5 Save Plot

### Save figures

```
>>> plt.savefig('foo.png')
```

### Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

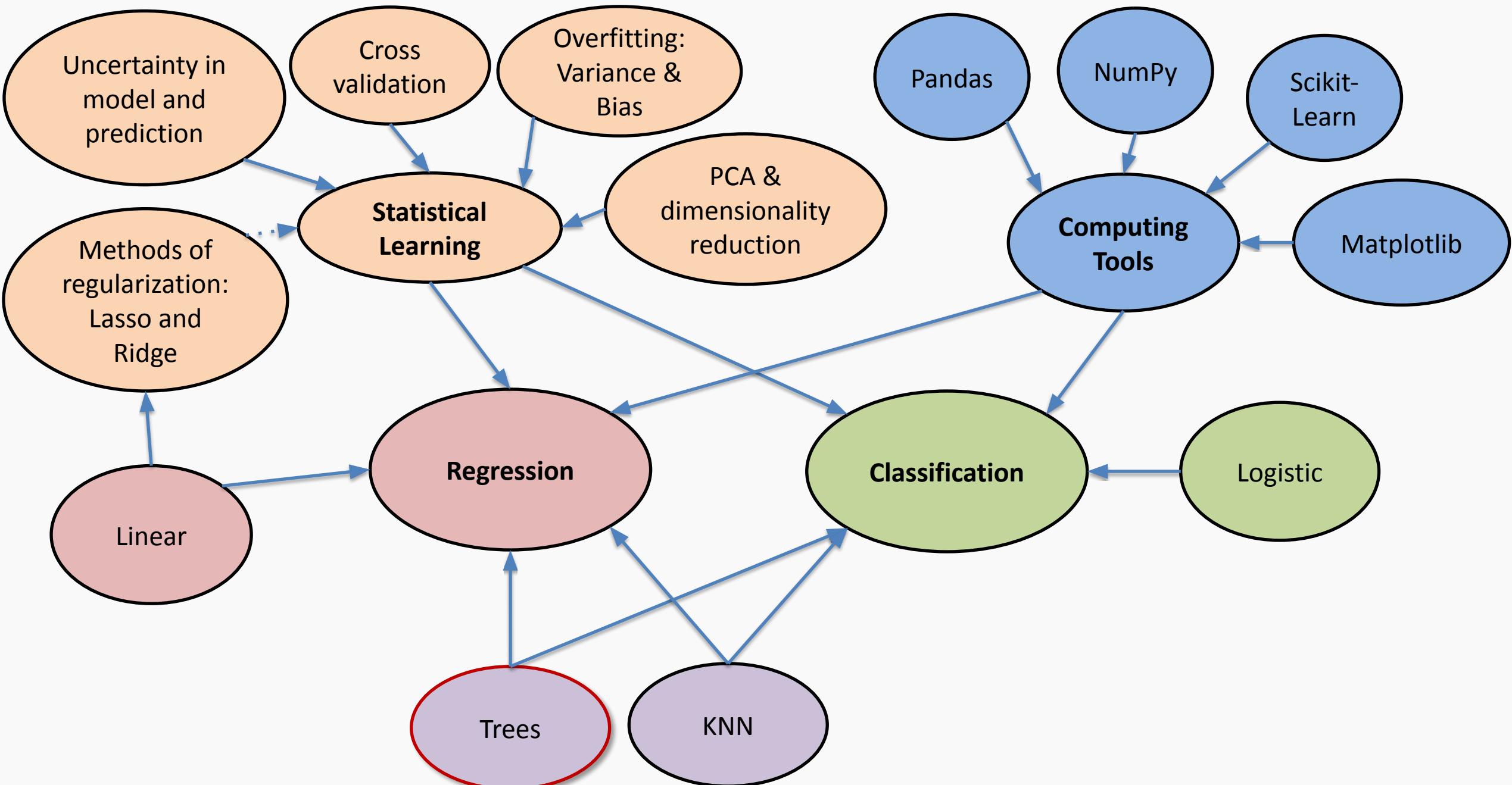
## 6 Show Plot

```
>>> plt.show()
```

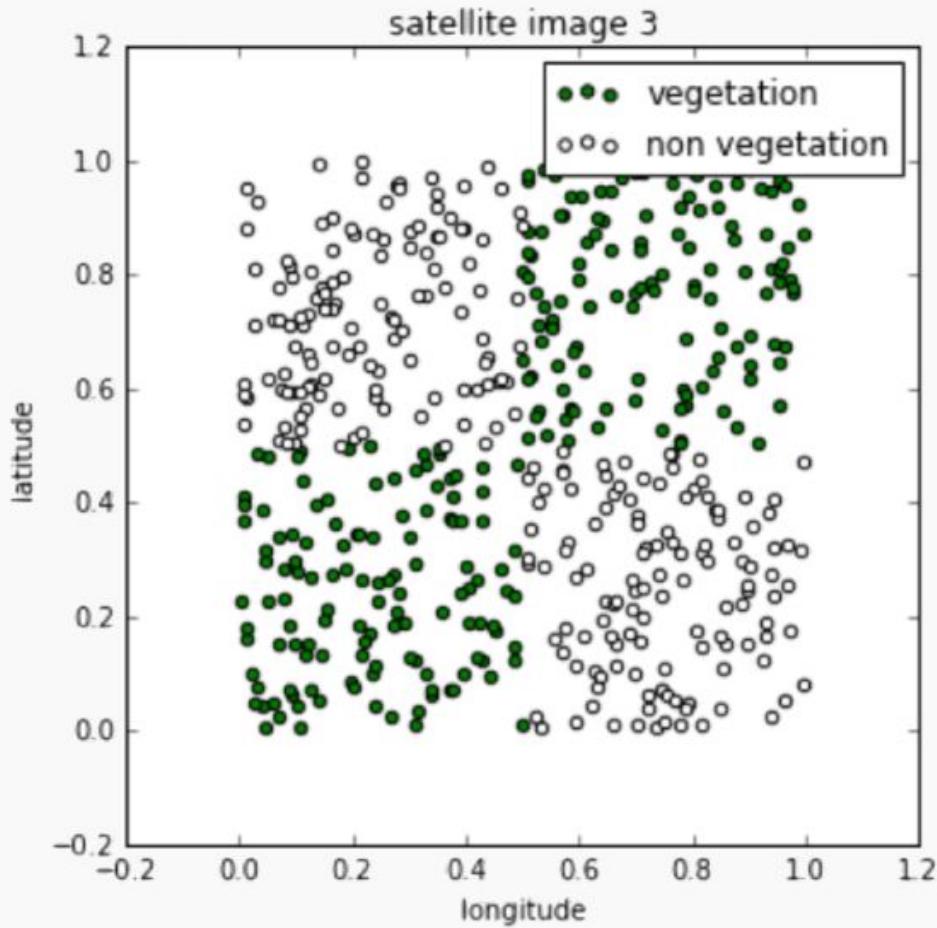
## Close & Clear

```
>>> plt.cla()
>>> plt.clf()
>>> plt.close()
```

Clear an axis  
Clear the entire figure  
Close a window



# SIMPLE DECISION TREE



Although **regression** models with linear boundaries are intuitive to interpret, it's harder to interpret non-linear decision boundaries.

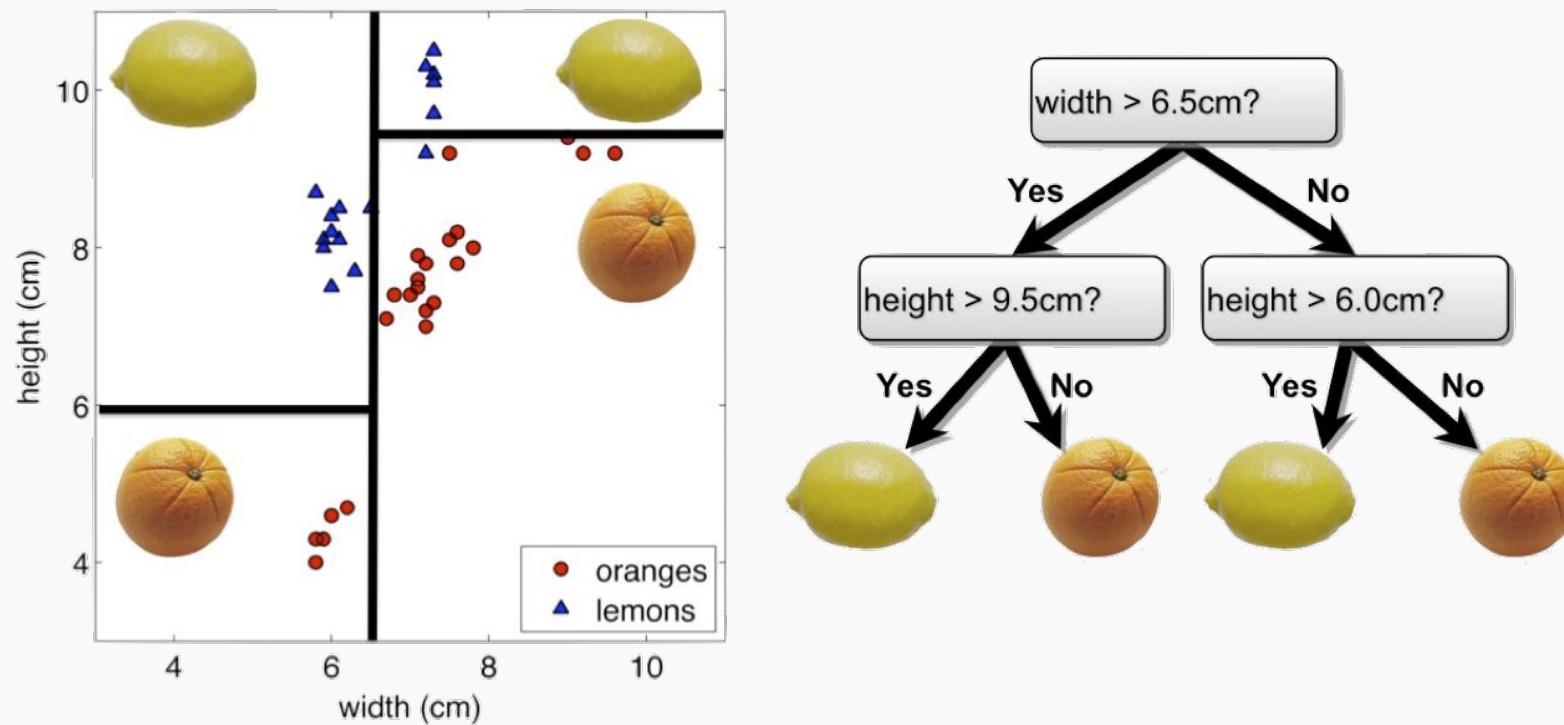
### Trees:

1. Allow for **complex decision boundaries**
2. Are easy to **interpret**

# The Geometry of Flow Charts

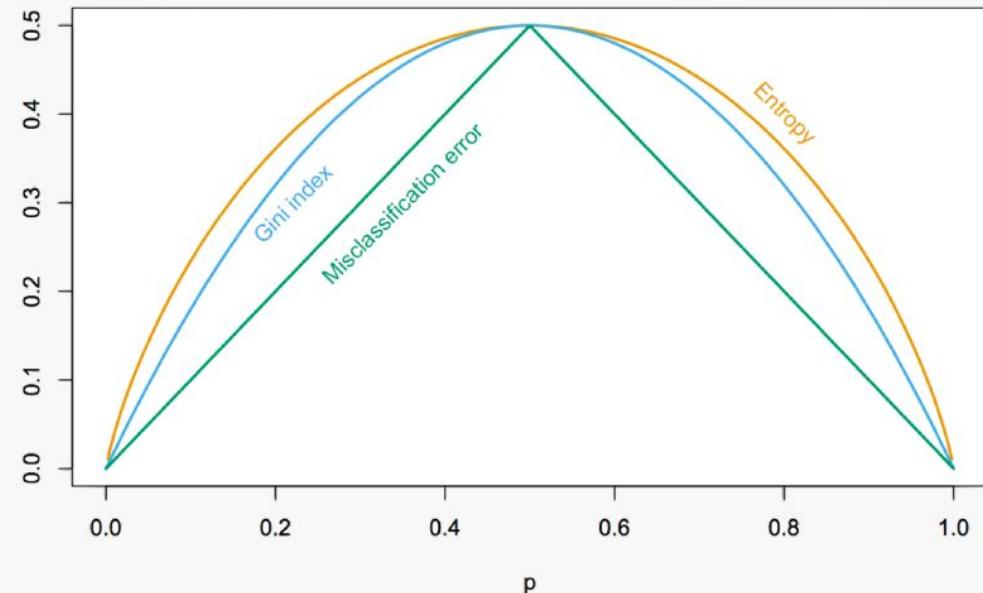
Each comparison and branching represents splitting a region in the feature space on a single feature.

The prediction is based on the most common class (or mean value).



# Considerations

1. Splitting Criterion. e.g.,
  - Gini Index
  - misclassification error
  - Entropy
  
2. Stopping Criterion. e.g.,
  - Minimum MSE
  - Uniformity of the data samples' labels
  - Size of tree, such as maximum depth
  - The “gain” converges



# Considerations

---

Shallow trees have: **high bias** and **low variance**

Deep trees have: **low bias** and **high variance**

Simple decision trees often:

- Overfit
- Underperform when compared to other classification and regression methods



# BAGGING



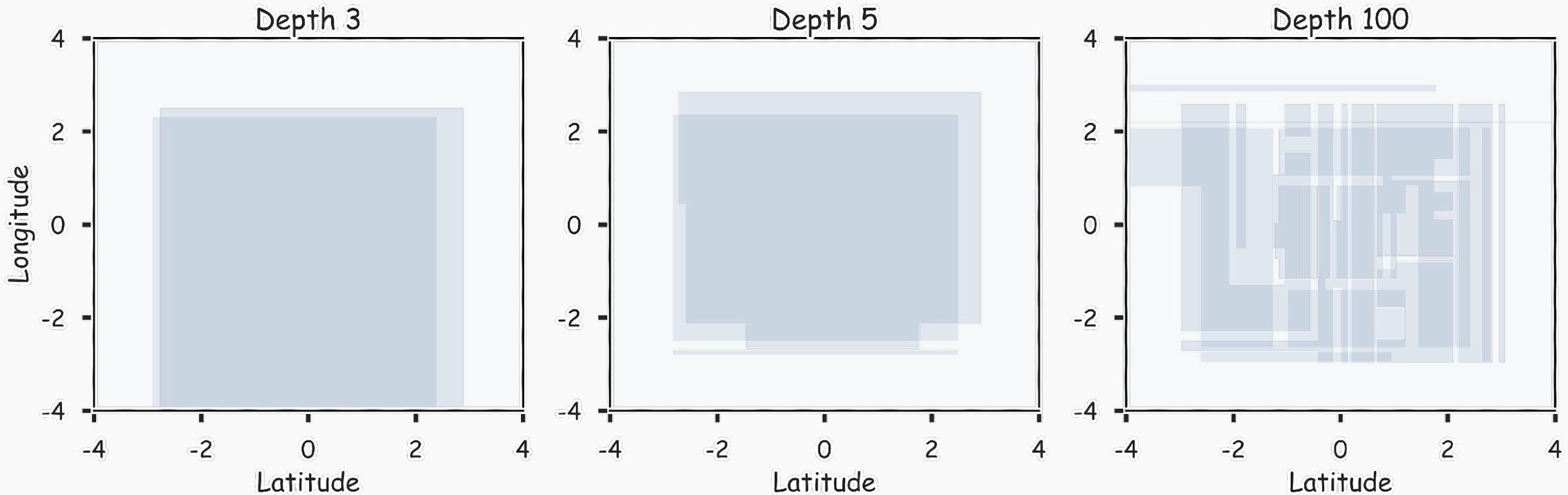
# Bootstrap Aggregating

---

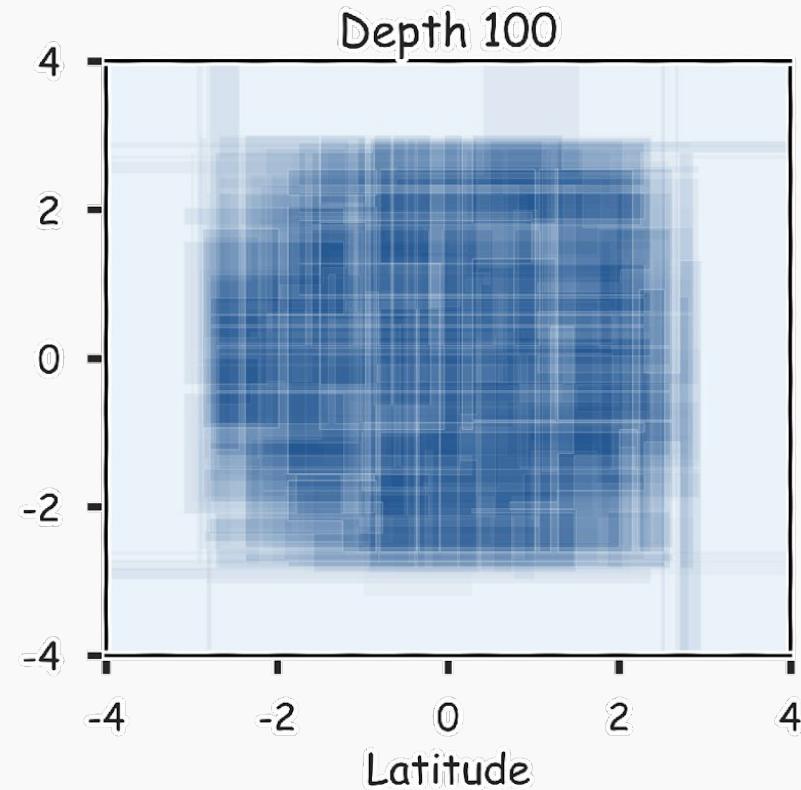
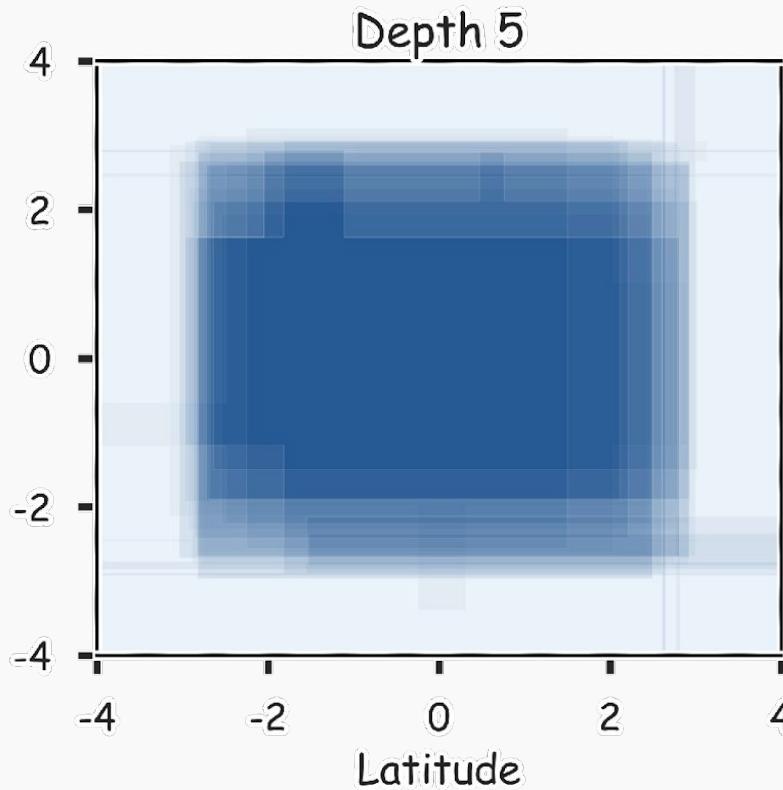
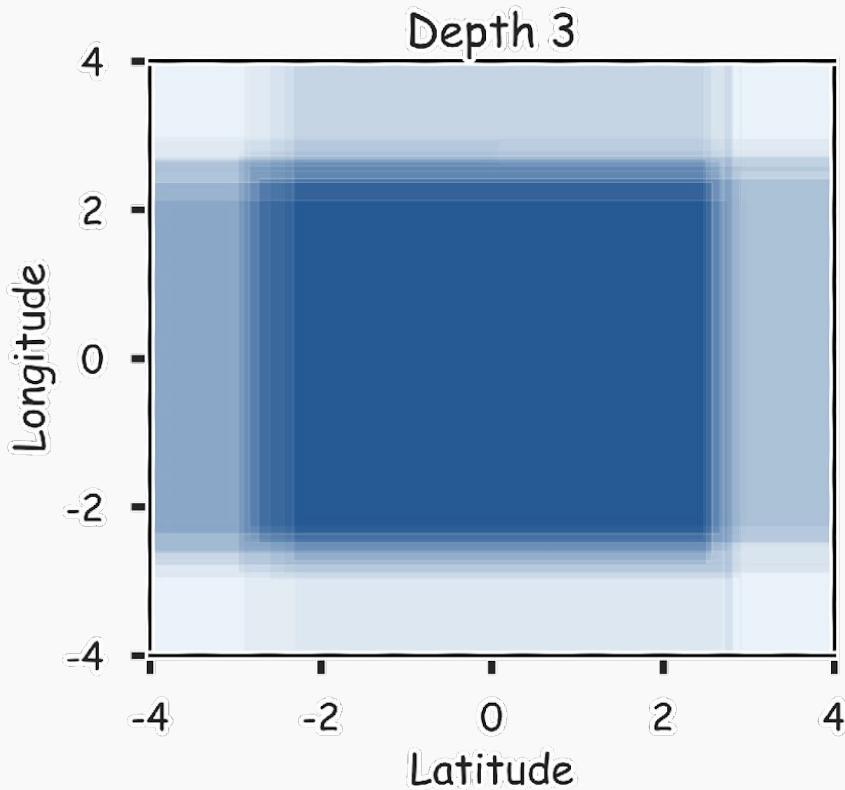
Bootstrap = generate data via sampling w/ replacement

Aggregating = return the average (regression) or majority class (classification)

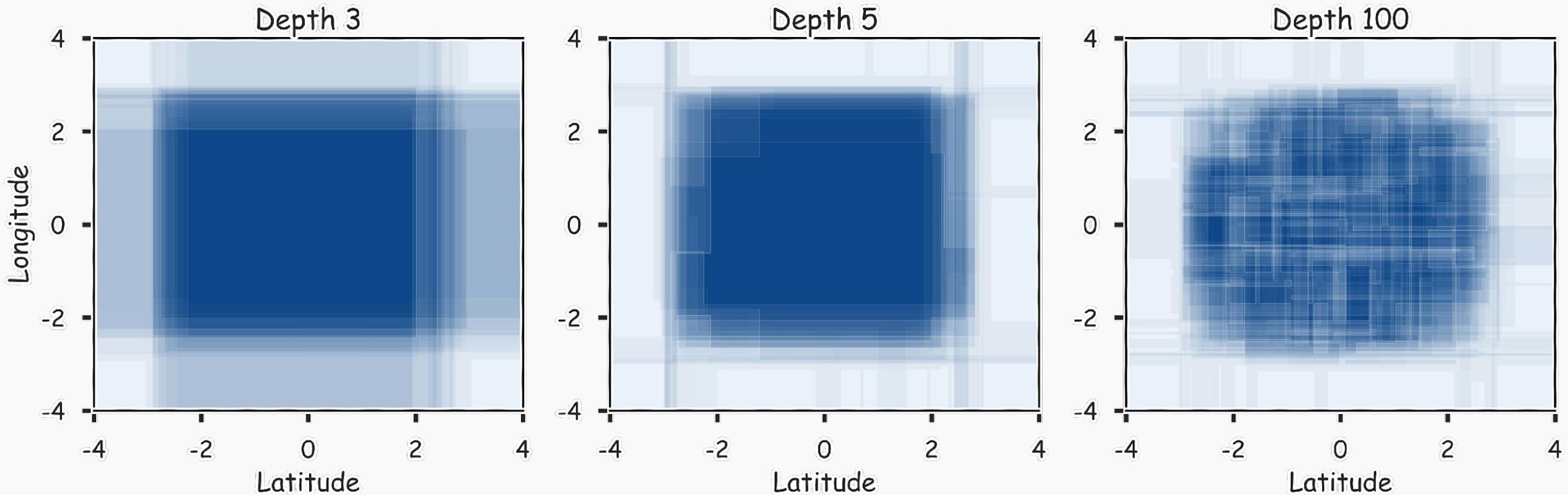
# Combine them? 2 magic realisms



# Combine them? 20 magic realisms

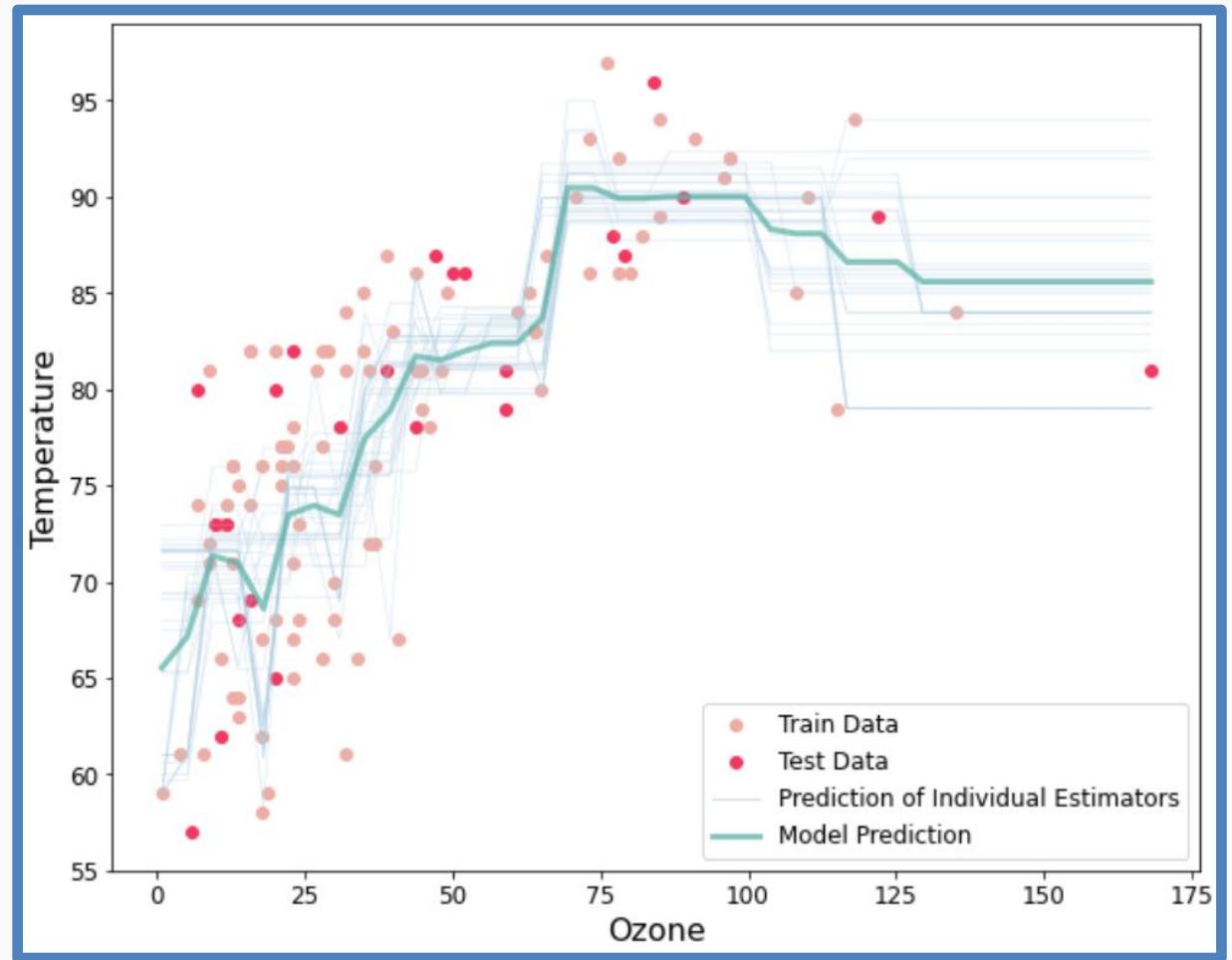


# Combine them? 100 magic realisms



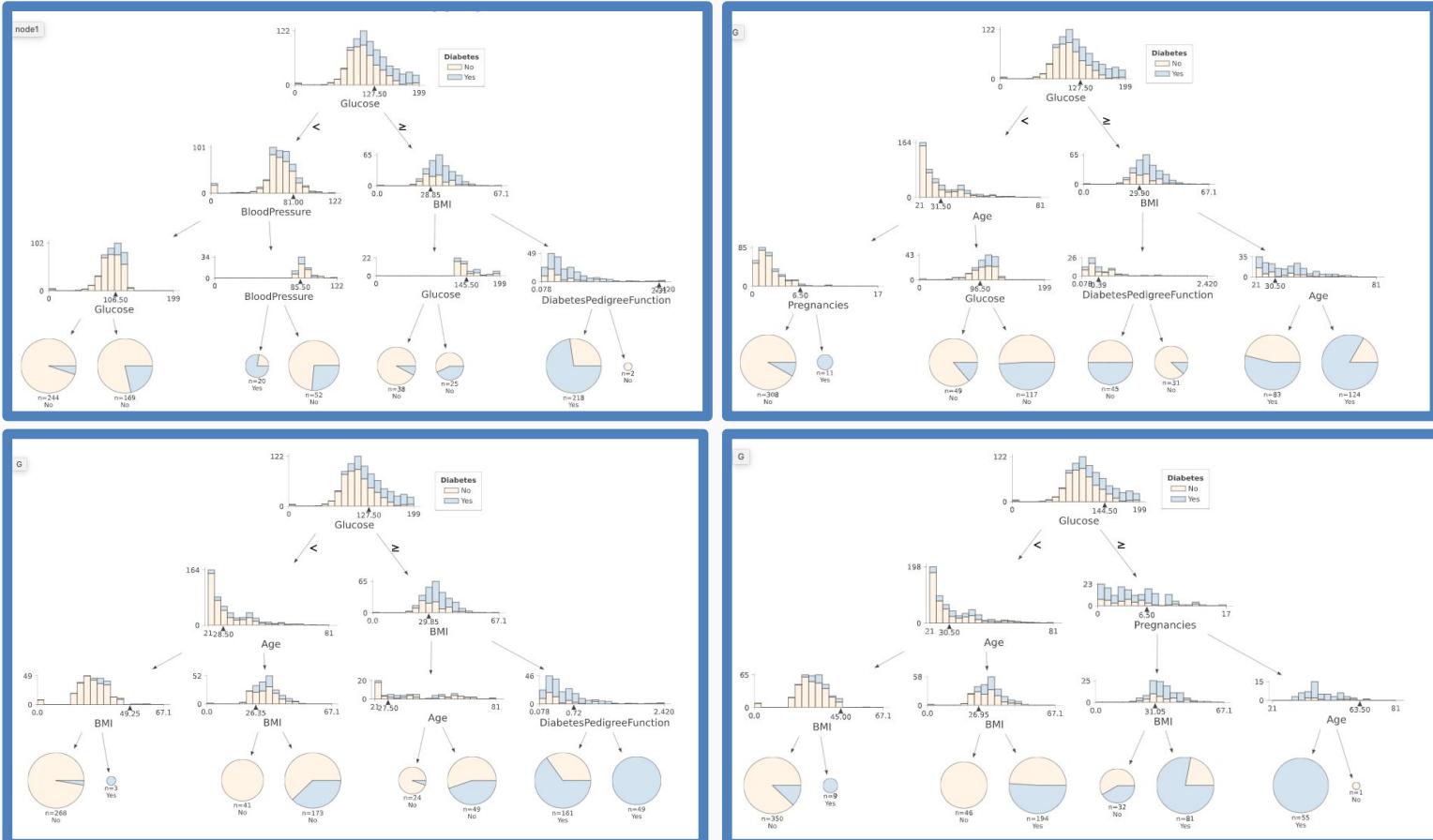
# Bagging (regression)

The resulting tree is the average of all tree (estimators).



# Bagging (classification)

For each bootstrap, we build a decision tree. The results is a combination (majority) of the predictions from all trees.



# Bootstrap Aggregating

## BENEFITS

- More expressive
- Helps prevent overfitting
- Decreases variance  
(less sensitive to different data)

## ISSUES

- interpretability ("majority")  
**solution:** variable importance via the avg Gini/MSE for each feature
- can still underfit or overfit  
**solution:** validation via out-of-bag error
- Trees tend to be **highly correlated**  
(split the same at the beginning)  
**solution:** random forests

# RANDOM FORESTS



# Random Forests

---

**Random Forest** is a modified form of bagging that creates ensembles of independent decision trees.

To de-correlate the trees, we:

1. train each tree on a **separate bootstrap sample** of the full training set (same as in bagging)
2. for each tree, at each split, we **randomly** select a set of  $J'$  predictors from the full set of predictors.

From amongst the  $J'$  predictors, we select the optimal predictor and the optimal corresponding threshold for the split.

# Random Forests

---

## SPECIFY

- Number of trees (`n_estimators`)
- Number of predictors (`max_features`)

## CONSIDERATIONS

- Be careful w/ the # of predictors. If you select a small %, you'll have an ensemble of weak models
- A lot of hyperparameters. Vary all of them together.



# BOOSTING



# Motivation for Boosting

---

**Question:** Could we address the shortcomings of single decision trees models in some other way?

For example, rather than performing variance reduction on complex trees, can we decrease the bias of simple trees - make them more **expressive**?

Can we learn from our **mistakes**?

A solution to this problem, making an expressive model from simple trees, is another class of ensemble methods called **boosting**.

# Gradient Boosting

---

The key intuition behind boosting is that one can take an ensemble of simple models  $\{T_h\}_{h \in H}$  and **additively** combine them into a single, more complex model.

Each model  $T_h$  might be a poor fit for the data, but a **linear combination** of the ensemble:

$$T = \sum_h \lambda_h T_h$$

can be **expressive/flexible**.

# Gradient Boosting: the algorithm

---

**Gradient boosting** is a method for iteratively building a complex regression model  $T$  by adding simple models.

Each new simple model added to the ensemble **compensates** for the weaknesses of the current ensemble.

# Gradient Boosting: the algorithm

1. Fit a simple model  $T^{(0)}$  on the training data

$$\{(x_1, y_1), \dots, (x_N, y_N)\}$$

Set  $T \leftarrow T^{(0)}$ .

Compute the residuals  $\{r_1, \dots, r_N\}$  for  $T$ .

2. Fit a simple model,  $T^{(1)}$ , to the current **residuals**, i.e. train using

$$\{(x_1, \mathbf{r}_1), \dots, (x_N, \mathbf{r}_N)\}$$

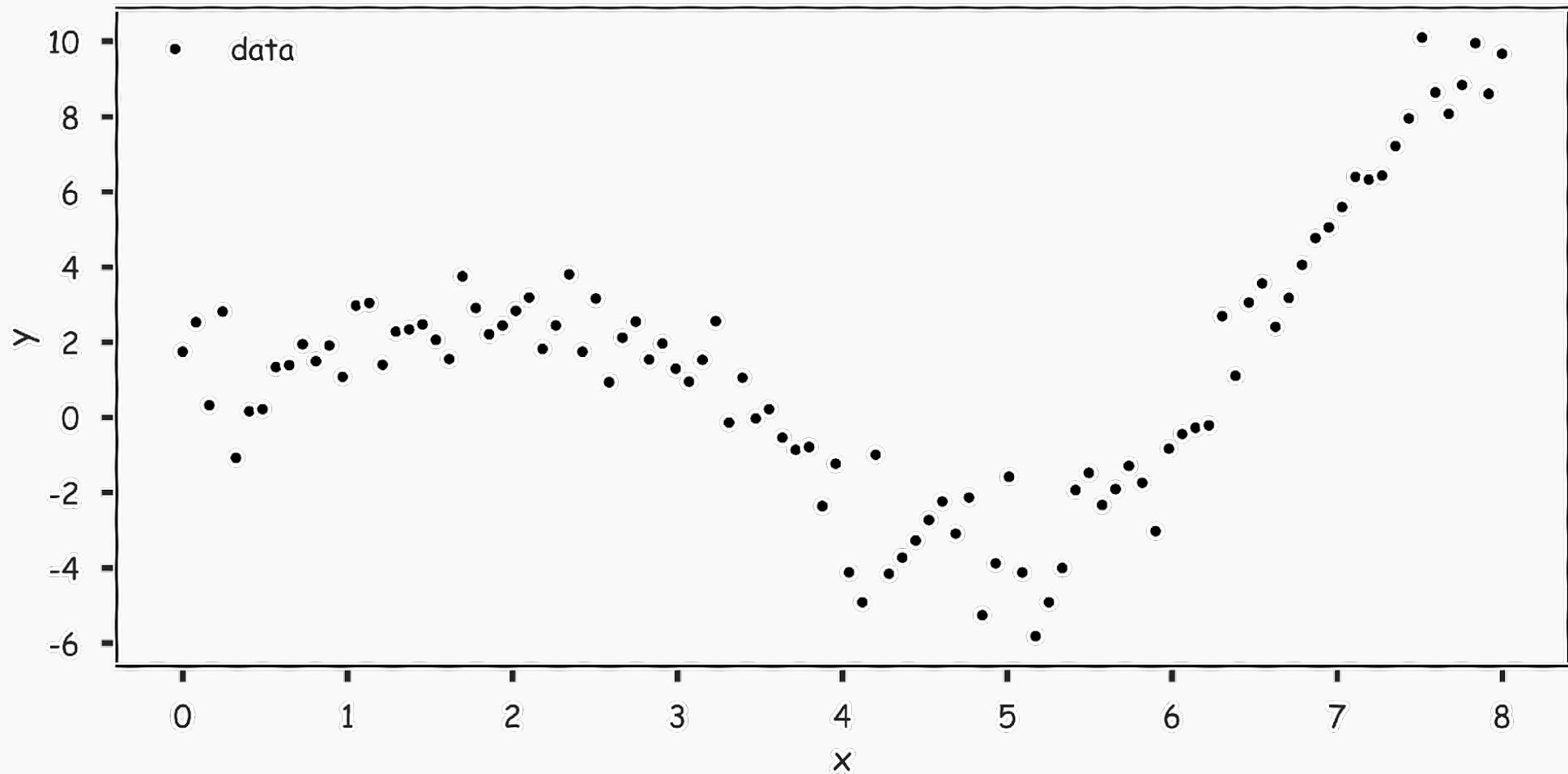
3. Set  $T \leftarrow T + \lambda T^{(1)}$

4. Compute residuals, set  $r_n \leftarrow r_n - \lambda T^i(x_n)$ ,  $n = 1, \dots, N$

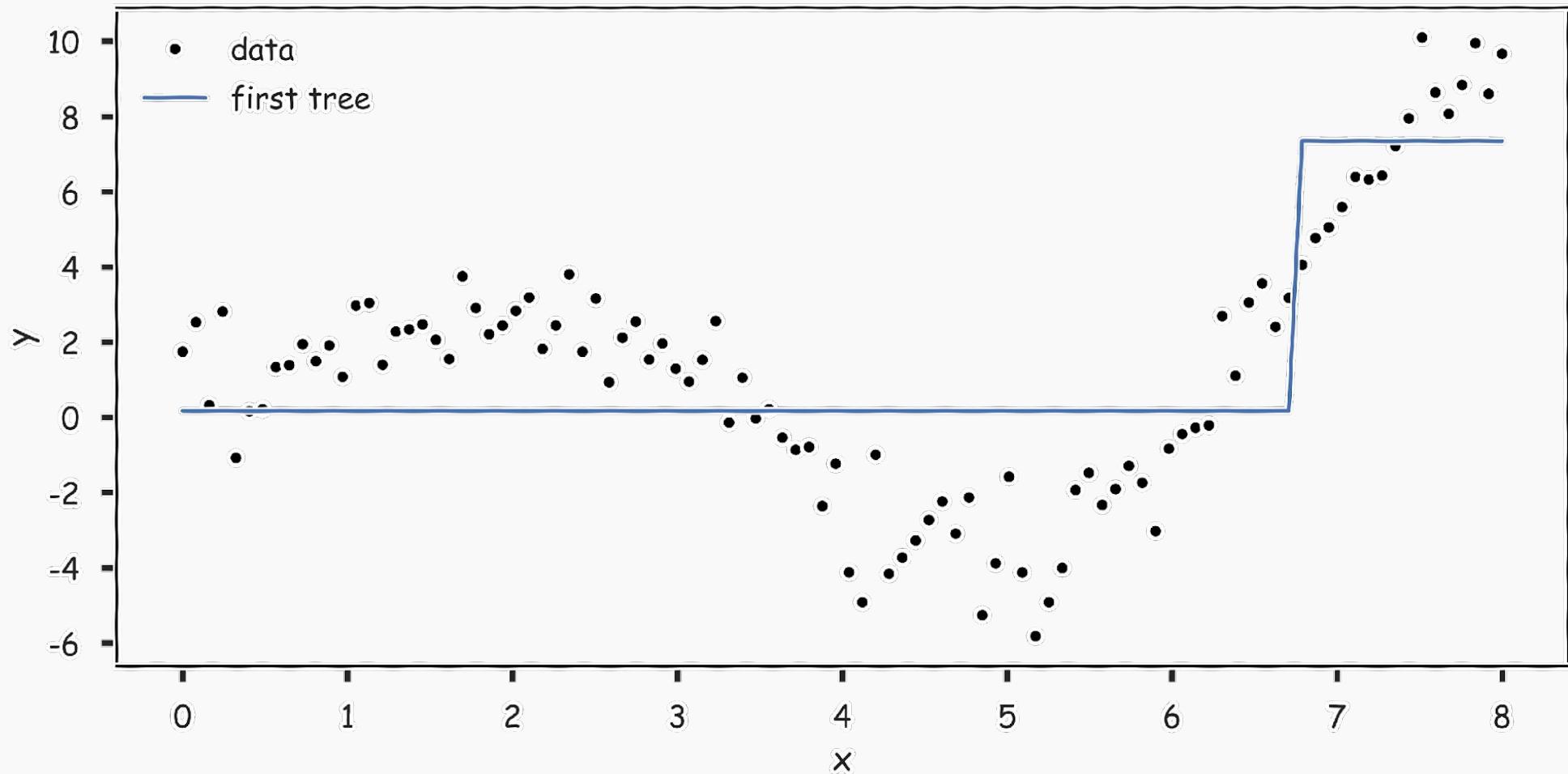
5. Repeat steps 2-4 until **stopping** condition met.

where  $\lambda$  is a constant called the **learning rate**.

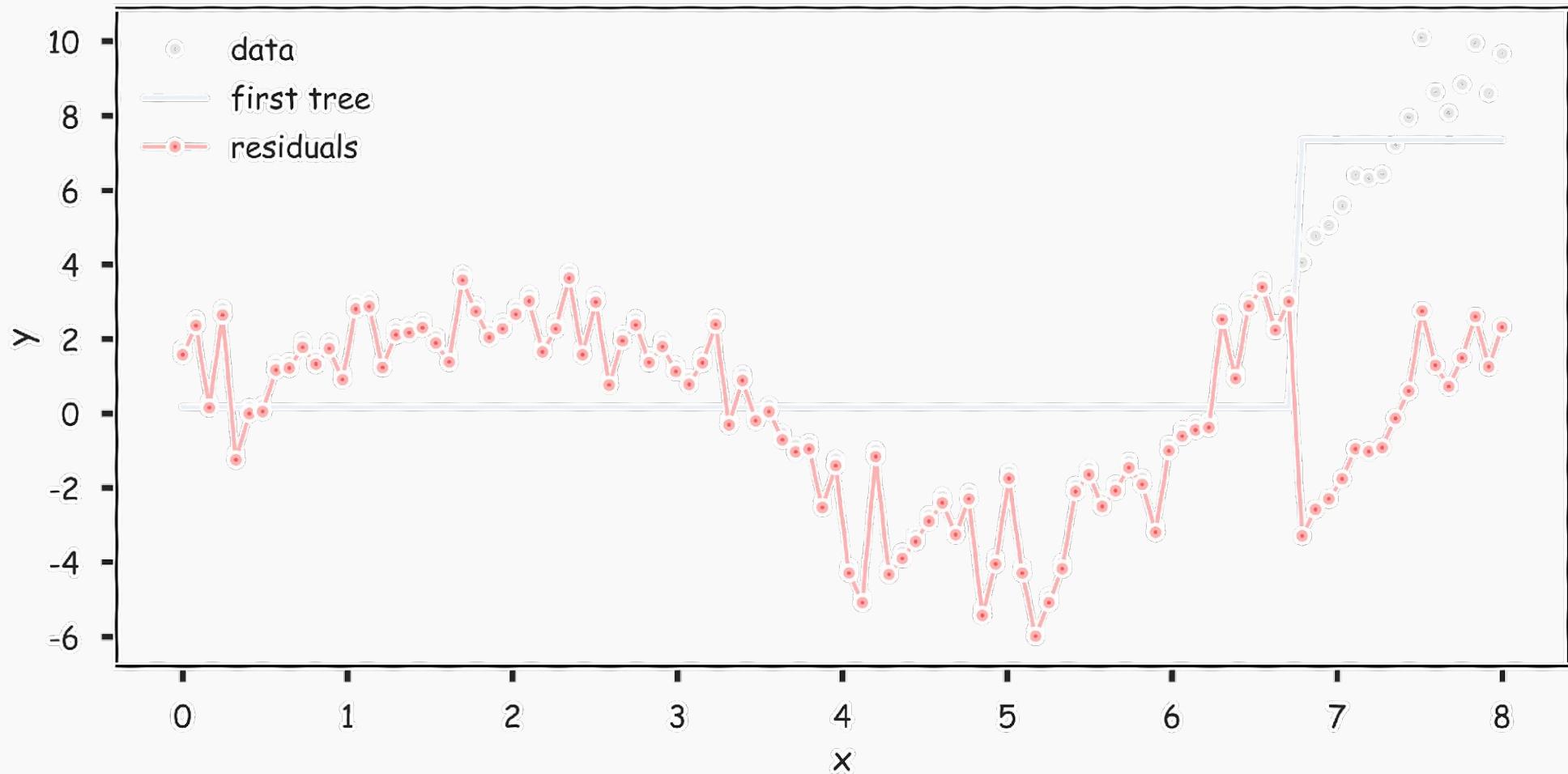
# Gradient Boosting: illustration



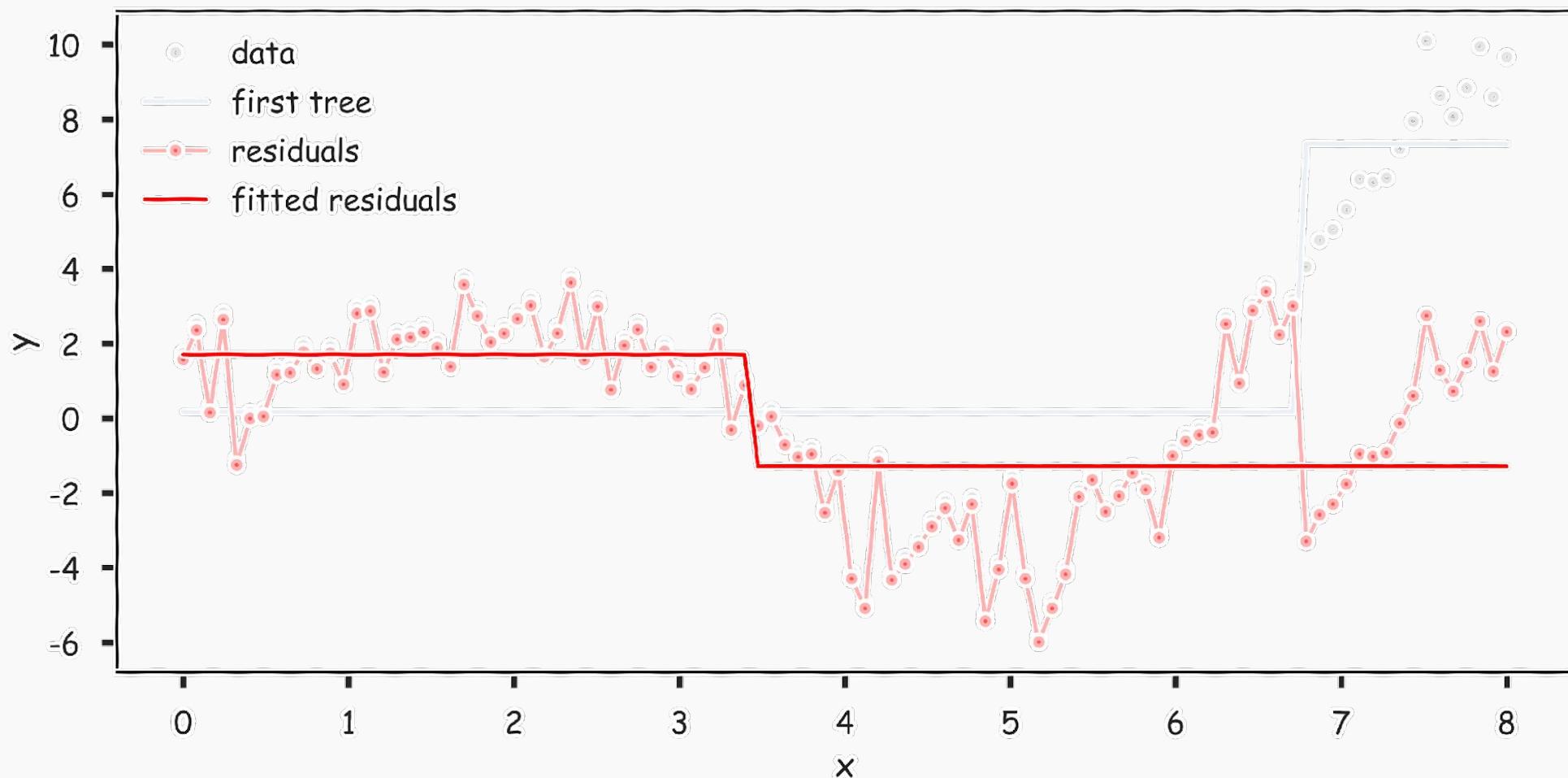
# Gradient Boosting: illustration



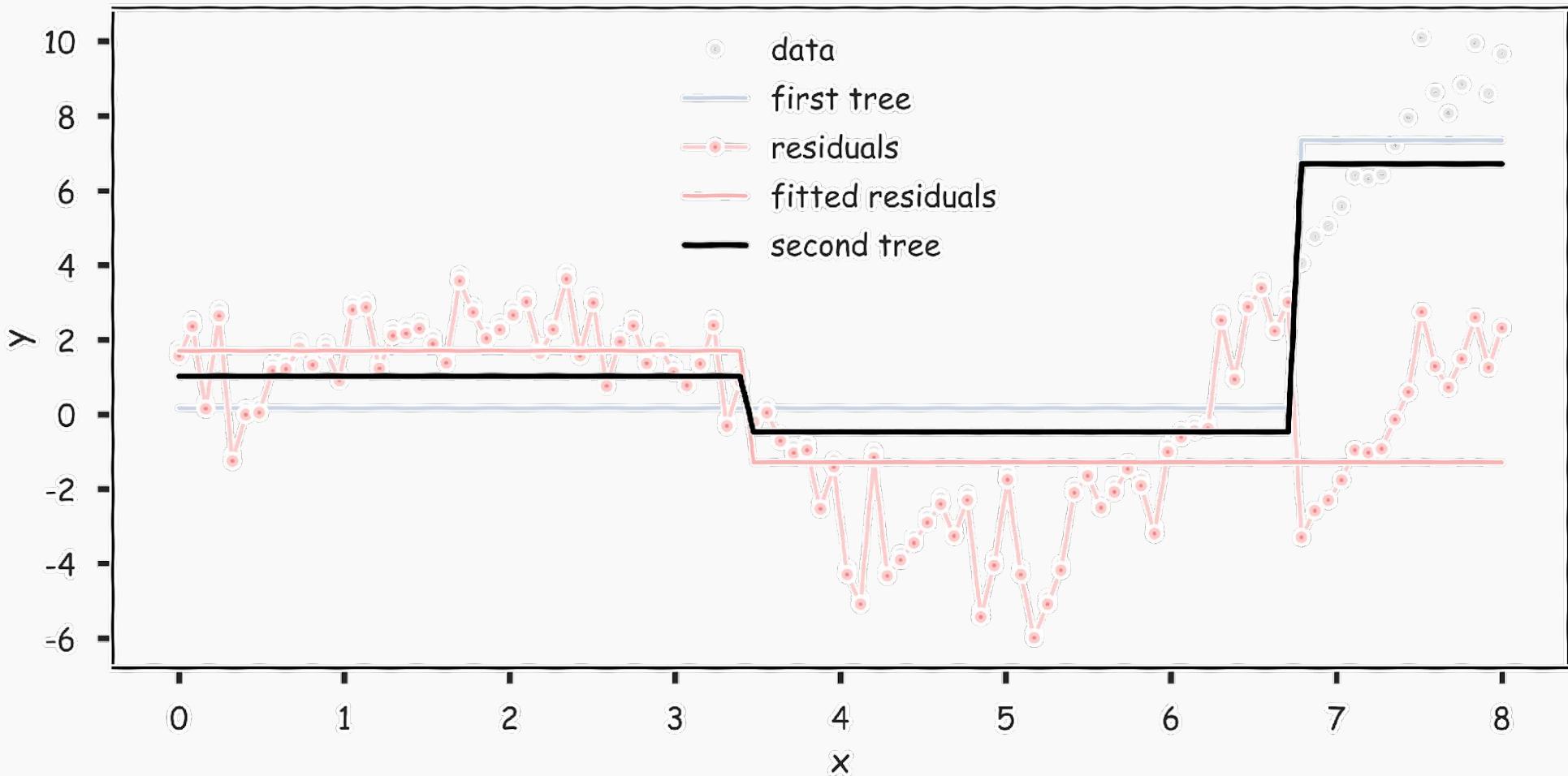
# Gradient Boosting: illustration



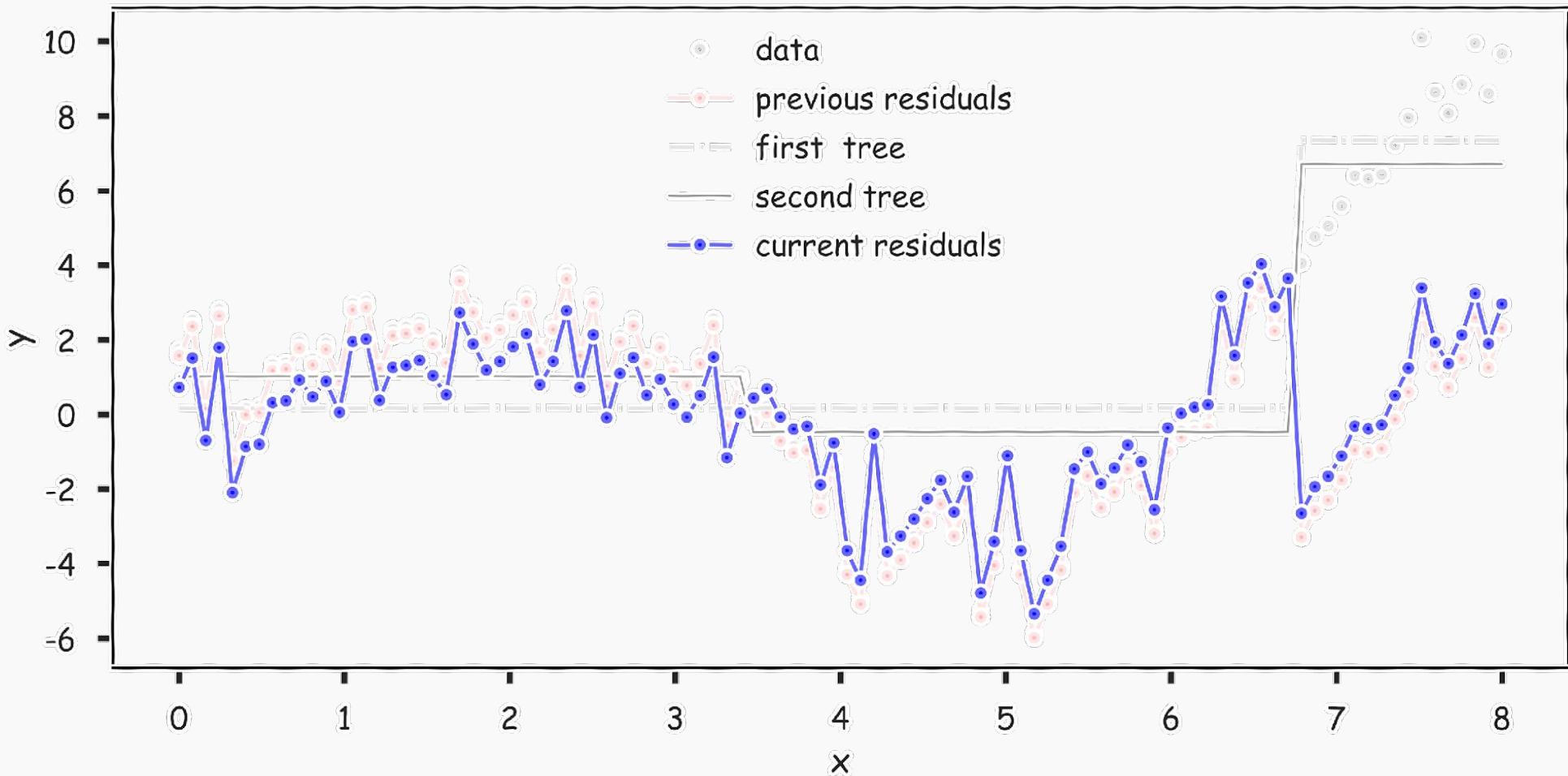
# Gradient Boosting: illustration



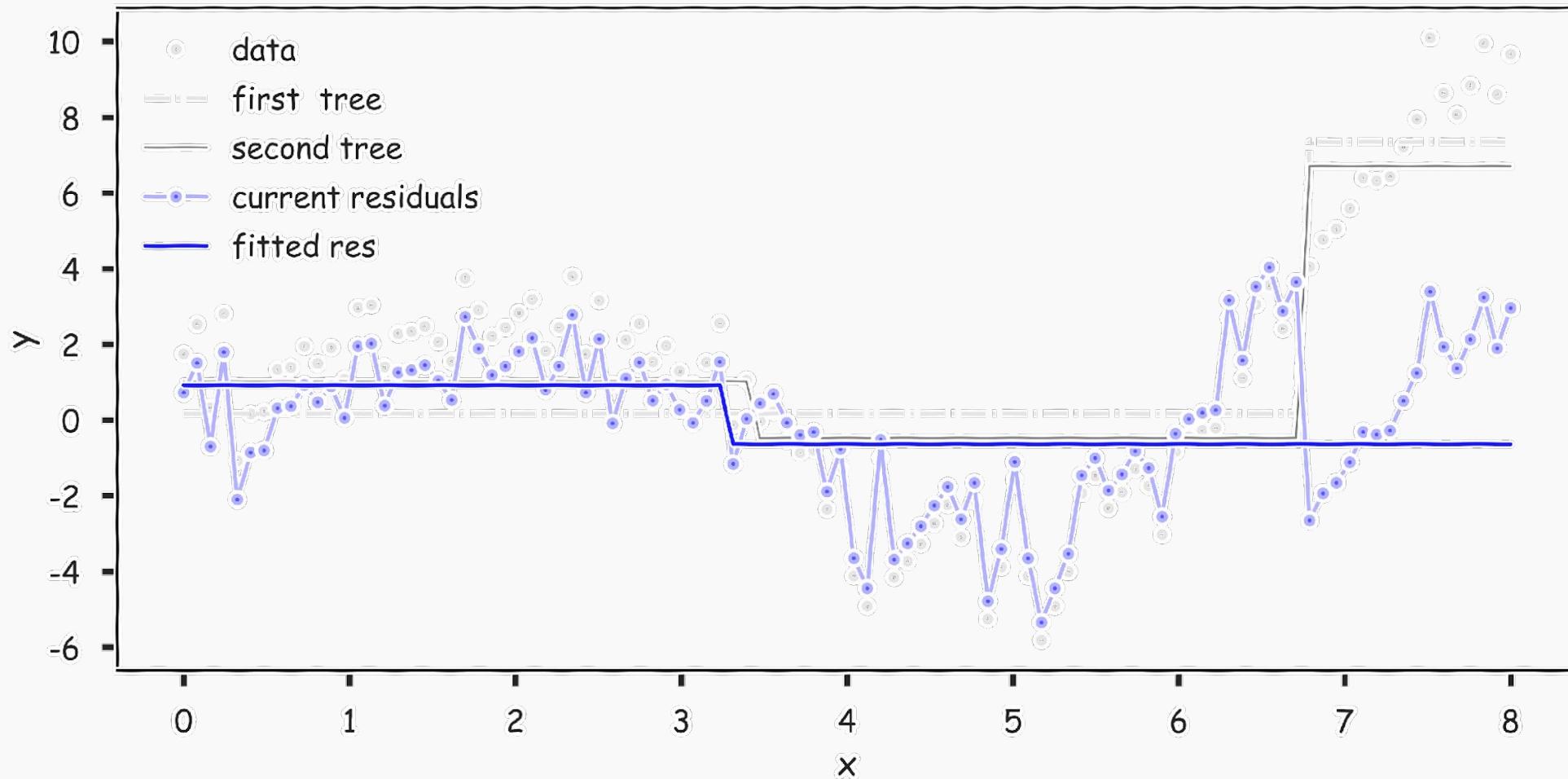
# Gradient Boosting: illustration



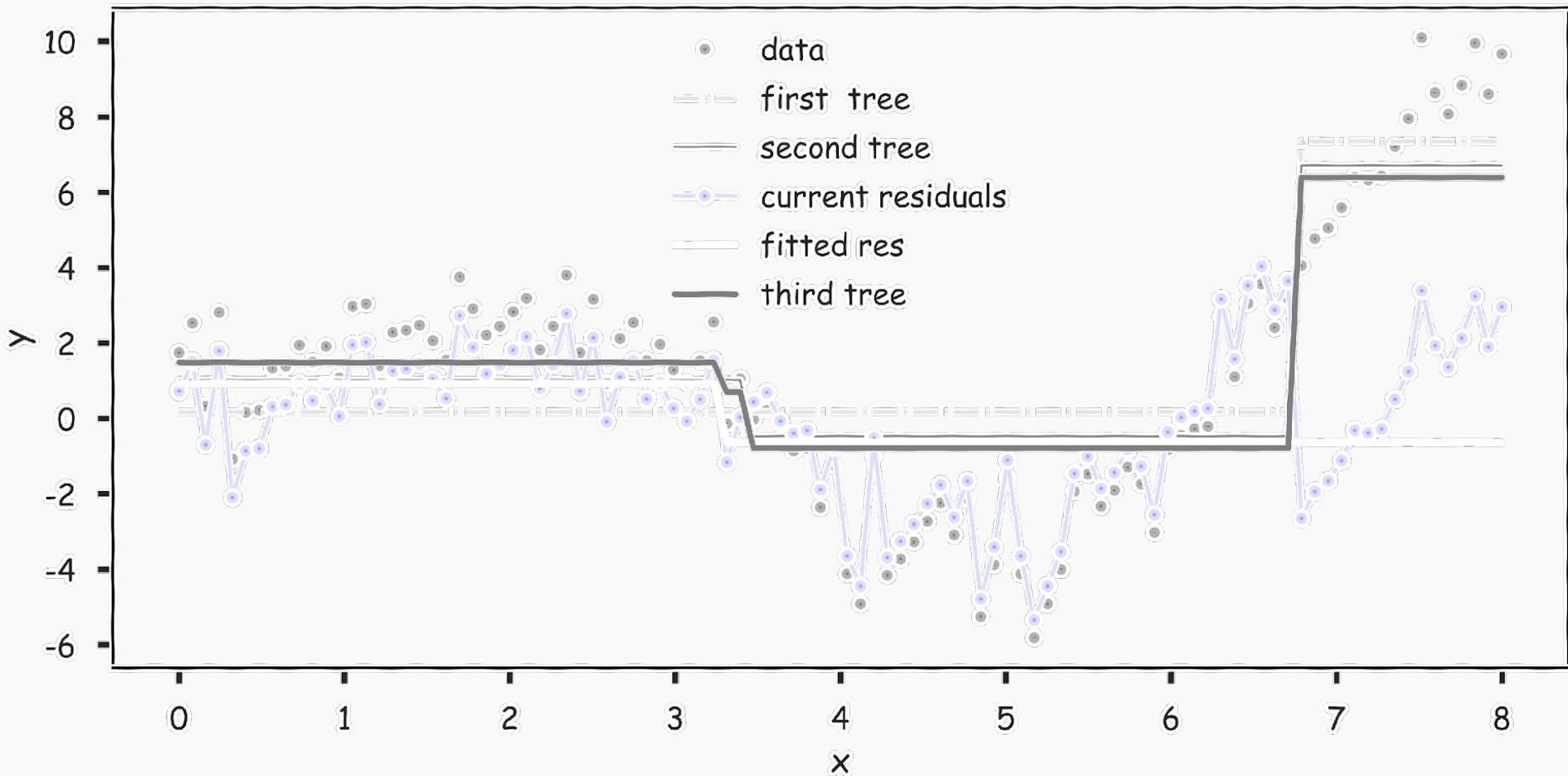
# Gradient Boosting: illustration

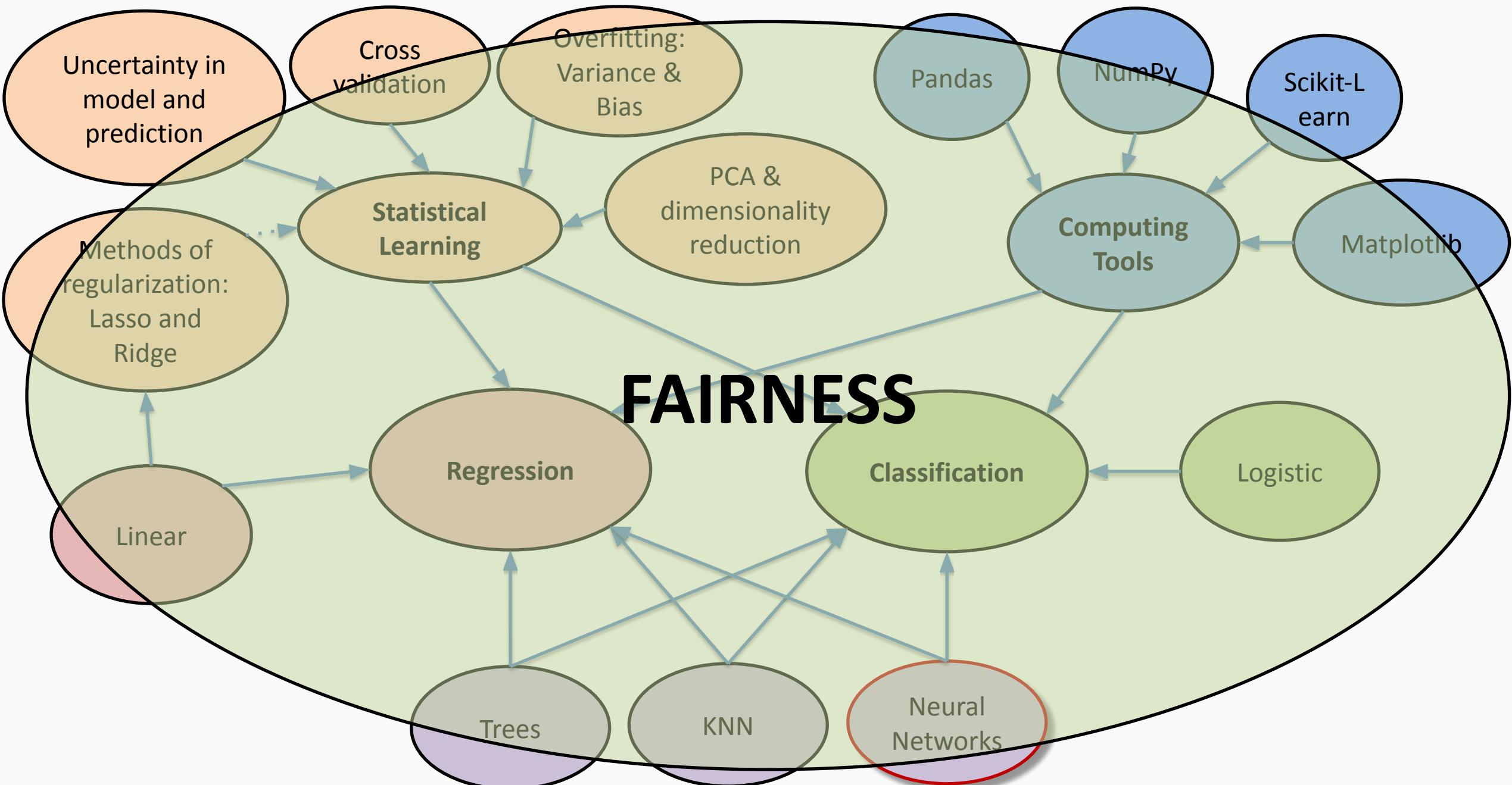


# Gradient Boosting: illustration



# Gradient Boosting: illustration





## THE PROBLEM OF UNFAIR BIAS

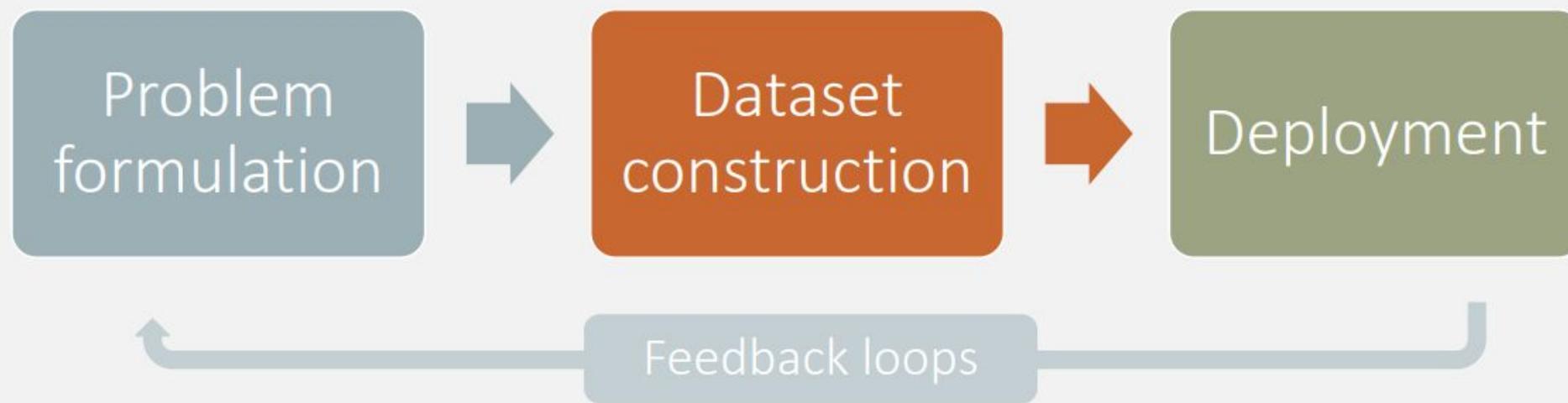
ML algorithms exhibit biases, and these biases often seem unfair

Examples:

- Amazon hiring
- PredPol predictive policing
- Skin-cancer detection



## HOW DOES BIAS ARISE?



## ASK QUESTIONS

Problem  
formulation

- Is a proprietary ML algorithm an ethical solution to the problem? Is this a decision for which the decision procedure should be transparent?
- Is it fair for this outcome to depend on this label?

## ASK QUESTIONS

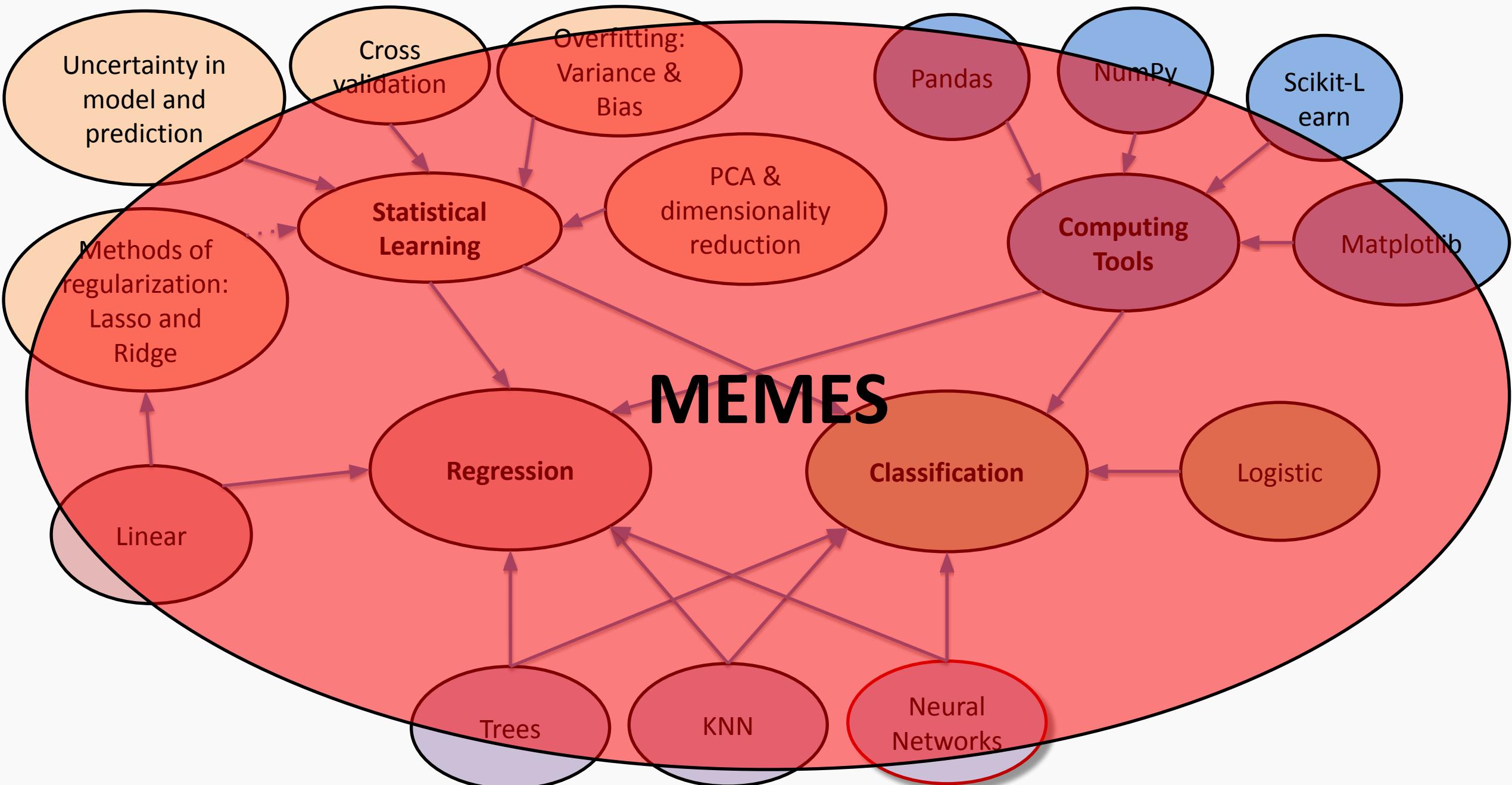
### Dataset construction

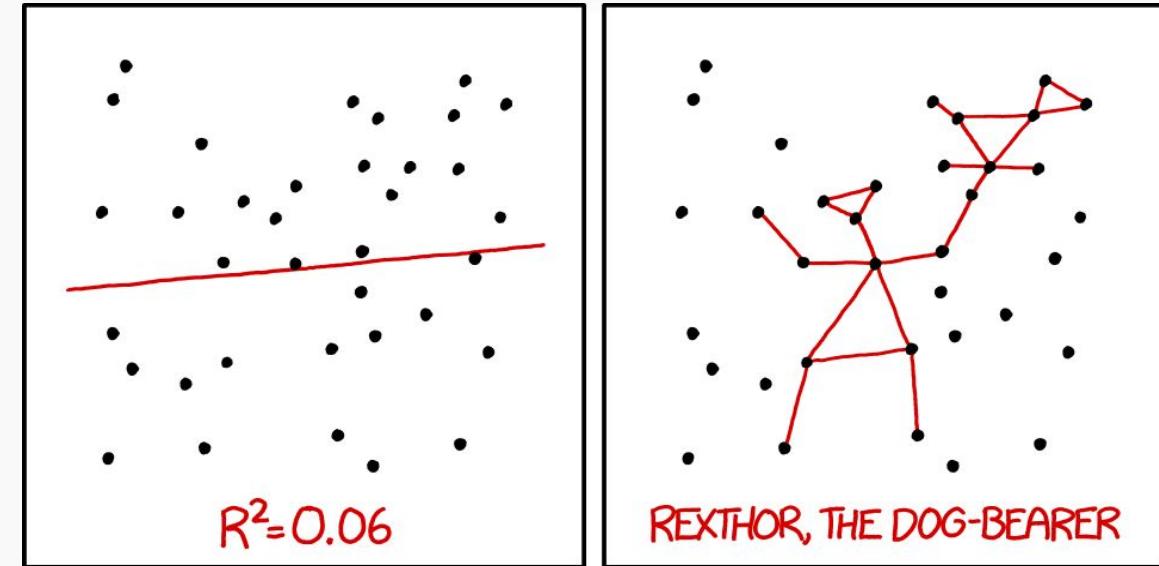
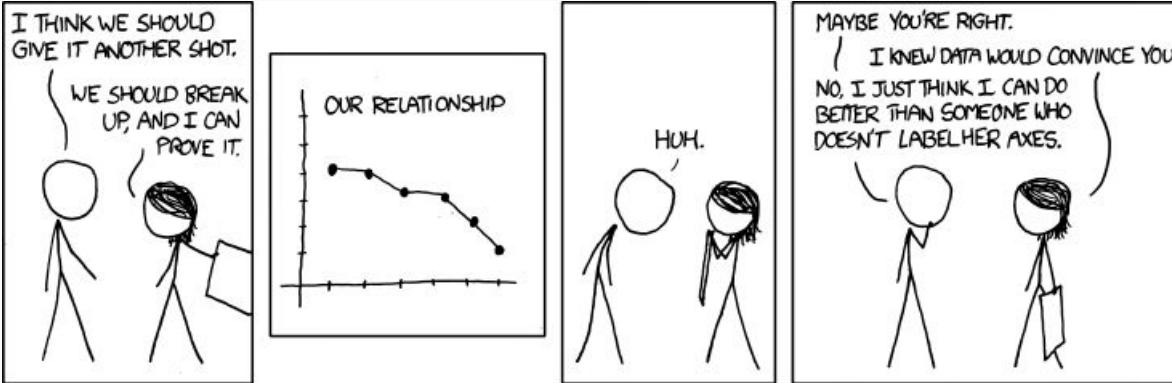
- Is the training dataset representative of the group on which the algorithm will be deployed?
- Do the training data reflect any unfair biases that will be reproduced by the algorithm?
- Is it fair for this outcome to depend on these predictors, given that such biases will be reproduced?

## ASK QUESTIONS

### Deployment

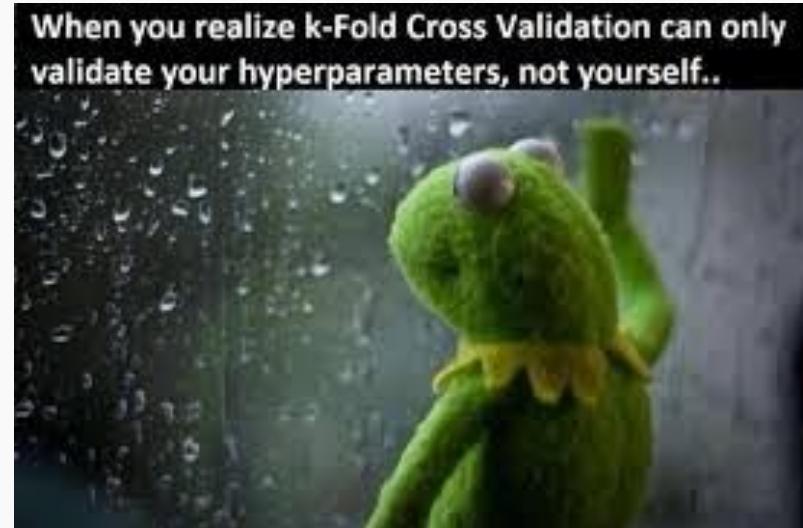
- Is the algorithm being deployed in a group that was adequately represented in the training data?
- What do users of the algorithm need to know to use it appropriately?



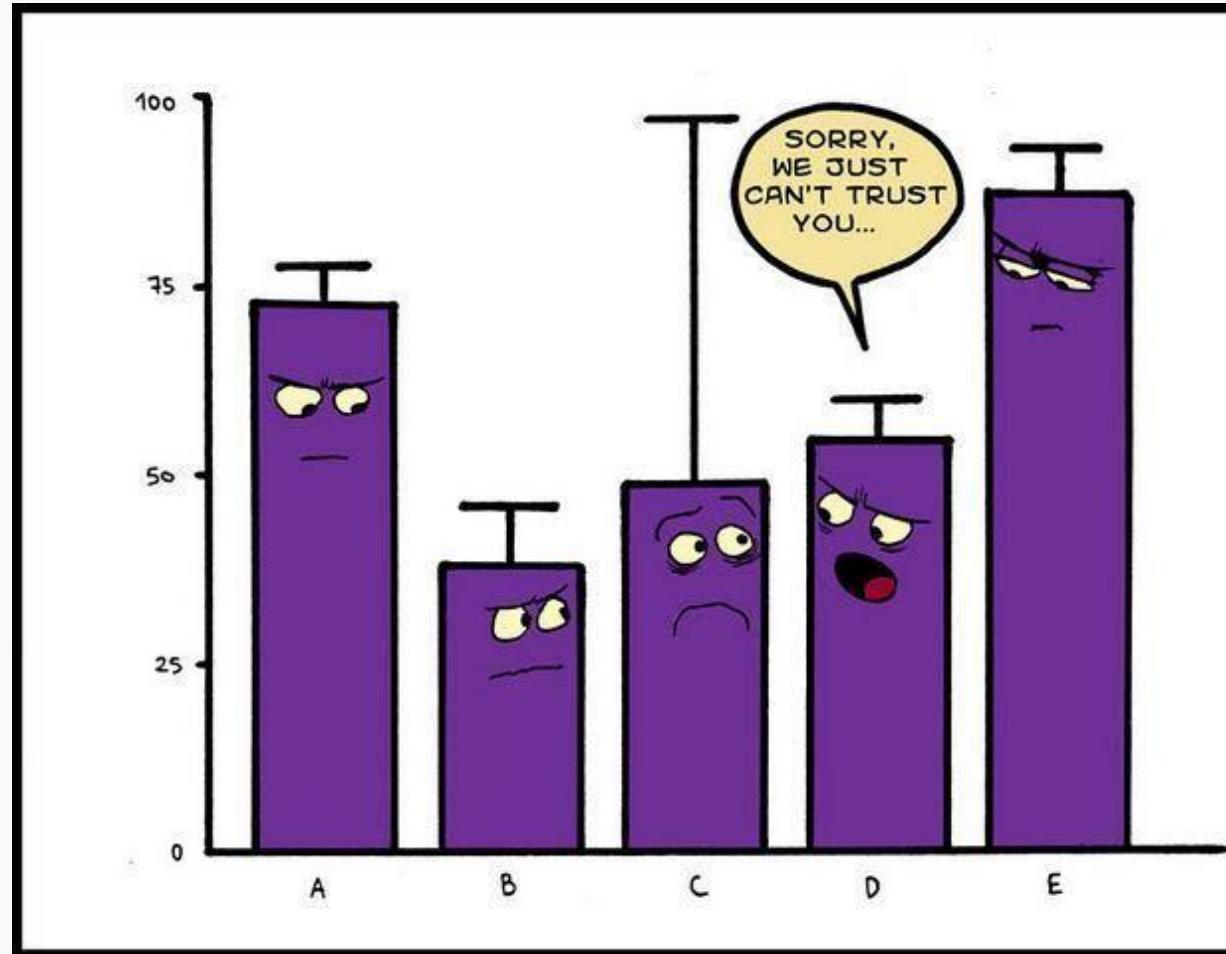


I DON'T TRUST LINEAR REGRESSIONS WHEN IT'S HARDER TO GUESS THE DIRECTION OF THE CORRELATION FROM THE SCATTER PLOT THAN TO FIND NEW CONSTELLATIONS ON IT.

When you realize k-Fold Cross Validation can only validate your hyperparameters, not yourself..



I finally remember what Zoom meetings remind me of.



WeKnowMemes

# GETTING VALUES FROM PANDAS

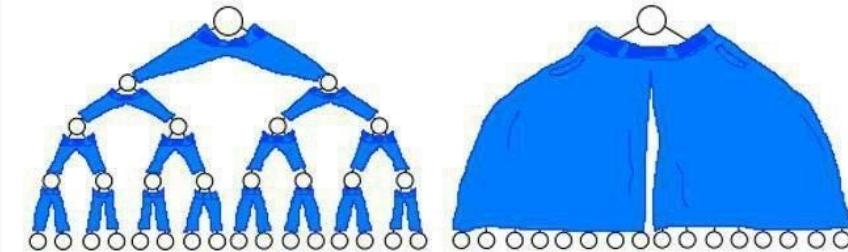


If a binary tree wore pants would he wear them

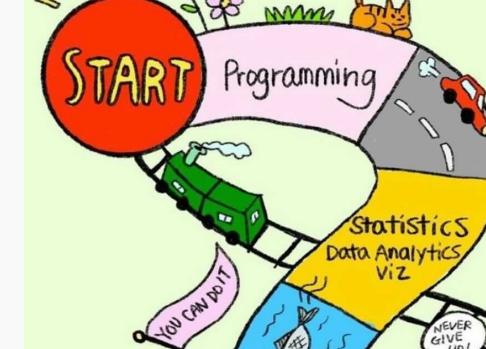
like this

or

like this?



**Path to Learn  
Data Science**



# Courses Related to Data Science

---

- **CS 109B: Advanced Topics in Data Science**
  - <https://harvard-iacs.github.io/2021-CS109B/>
- CS 171: Visualizations
- CS 181/281: Machine Learning
- CS 182: Artificial Intelligence (AI)
- CS 205: Distributive Computing
- Stat 110/210: Probability Theory
- Stat 111/211: Statistical Inference
- Stat 139: Linear Models
- Stat 149: Generalized Linear Models
- Stat 195: Statistical Machine Learning

This list is not exhaustive!



# State of Machine Learning and Data Science 2020

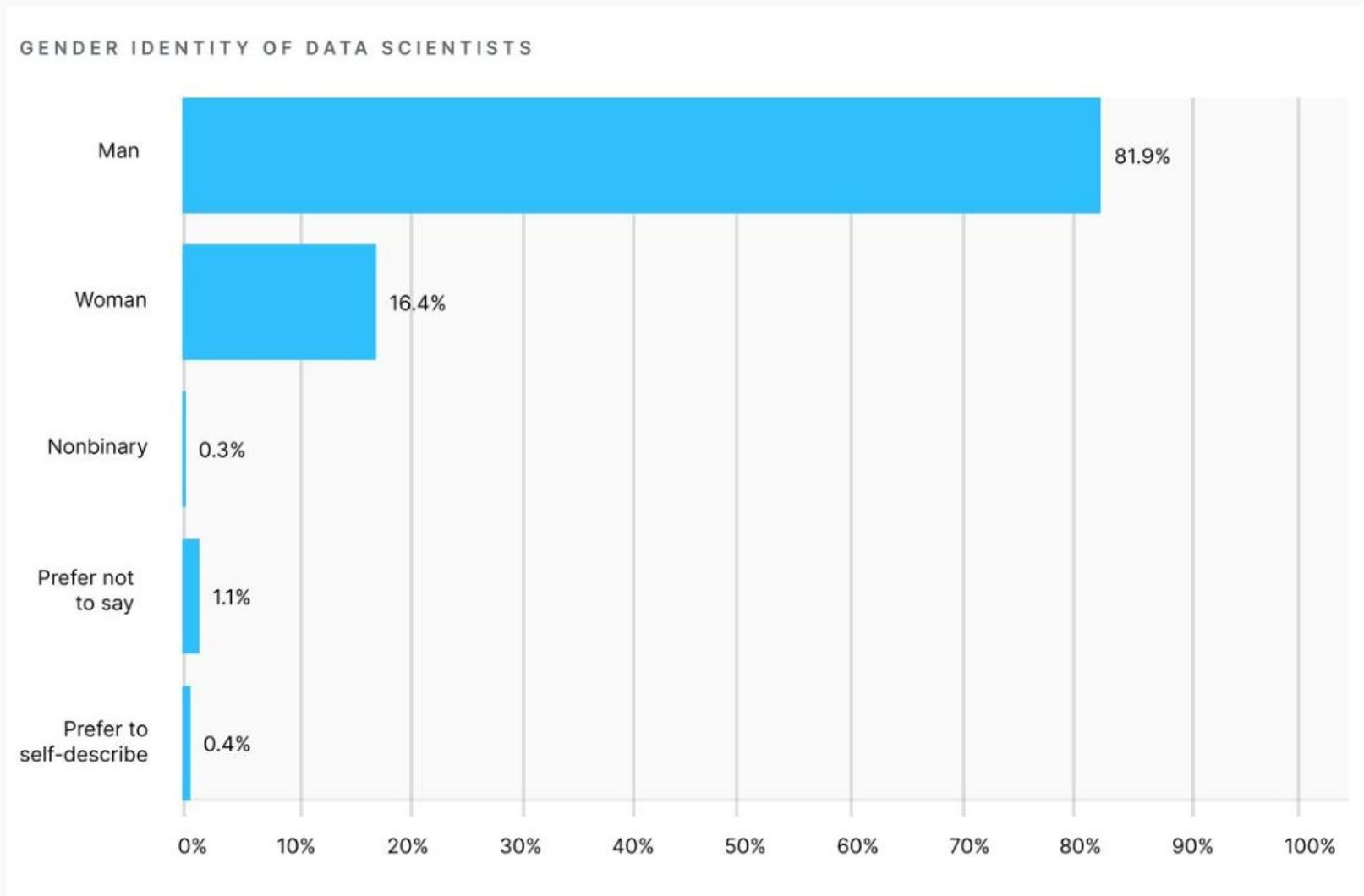
---

## Kaggle enterprise executive summary report

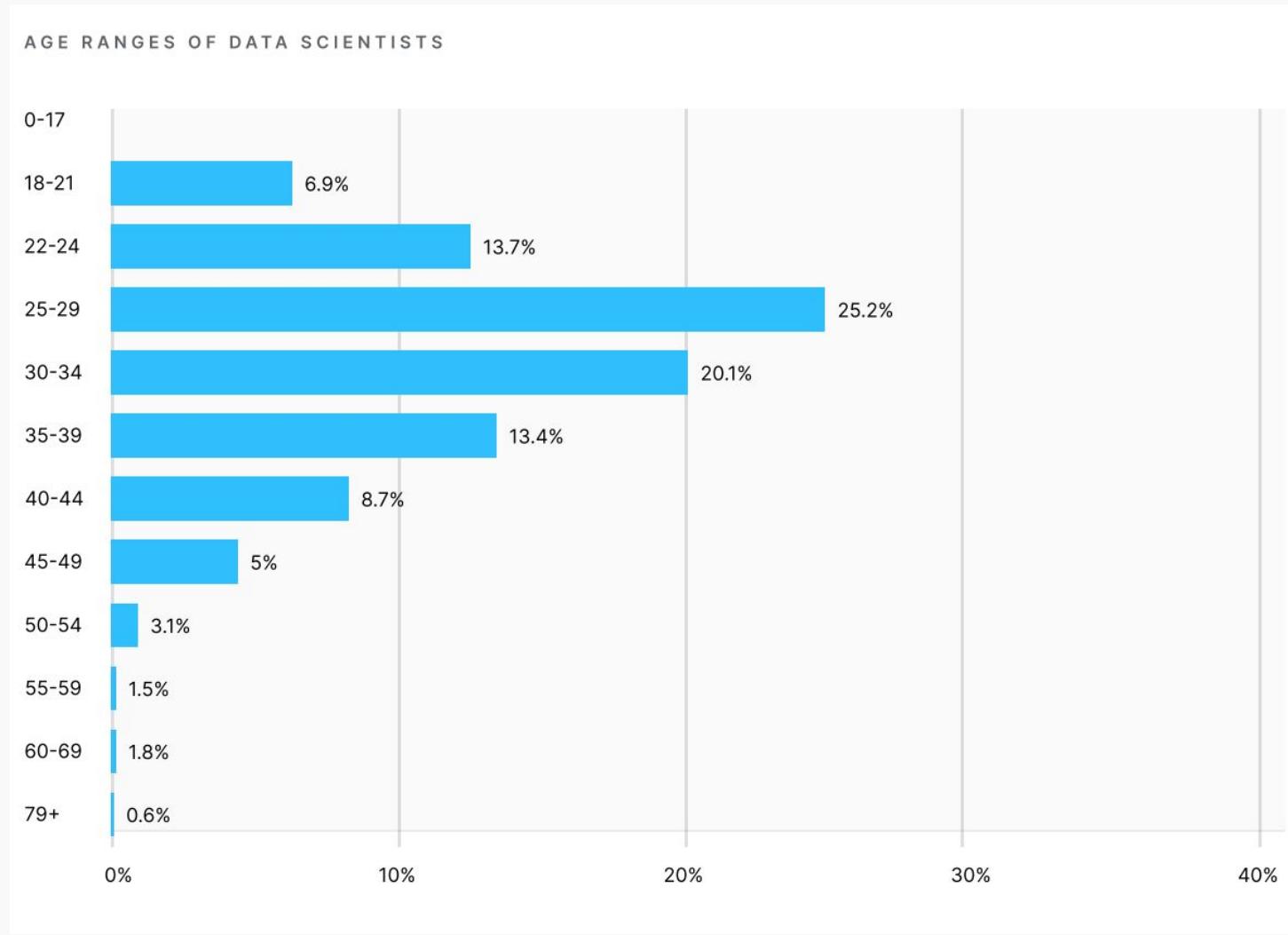
Kaggle surveyed its community of data enthusiasts to share trends within a quickly growing field.

Based on responses from 20,036 Kaggle members, they've created a report focused on the 13% (2,675 respondents) who are [currently employed](#) as data scientists.

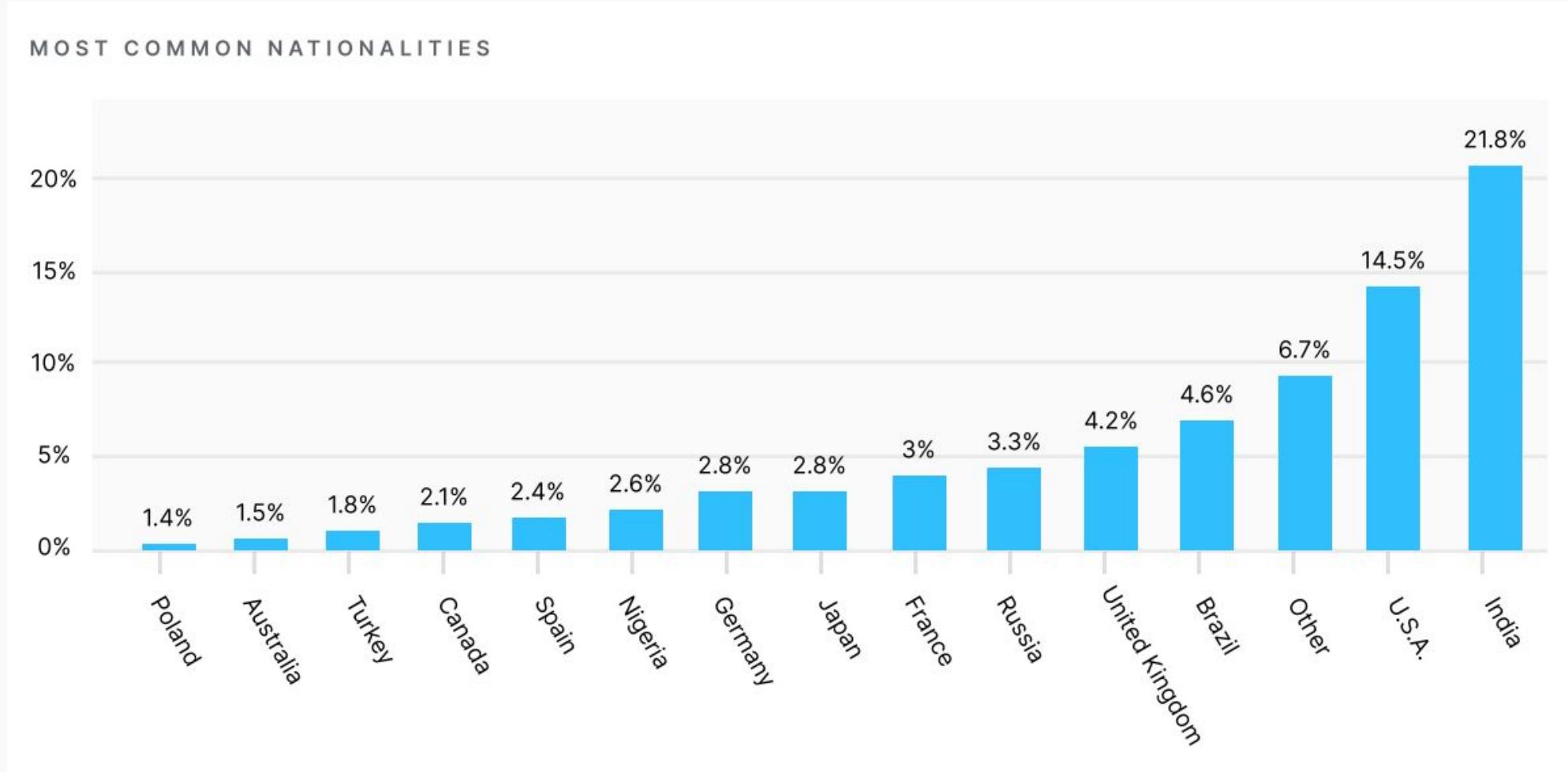
# Key findings: Gender



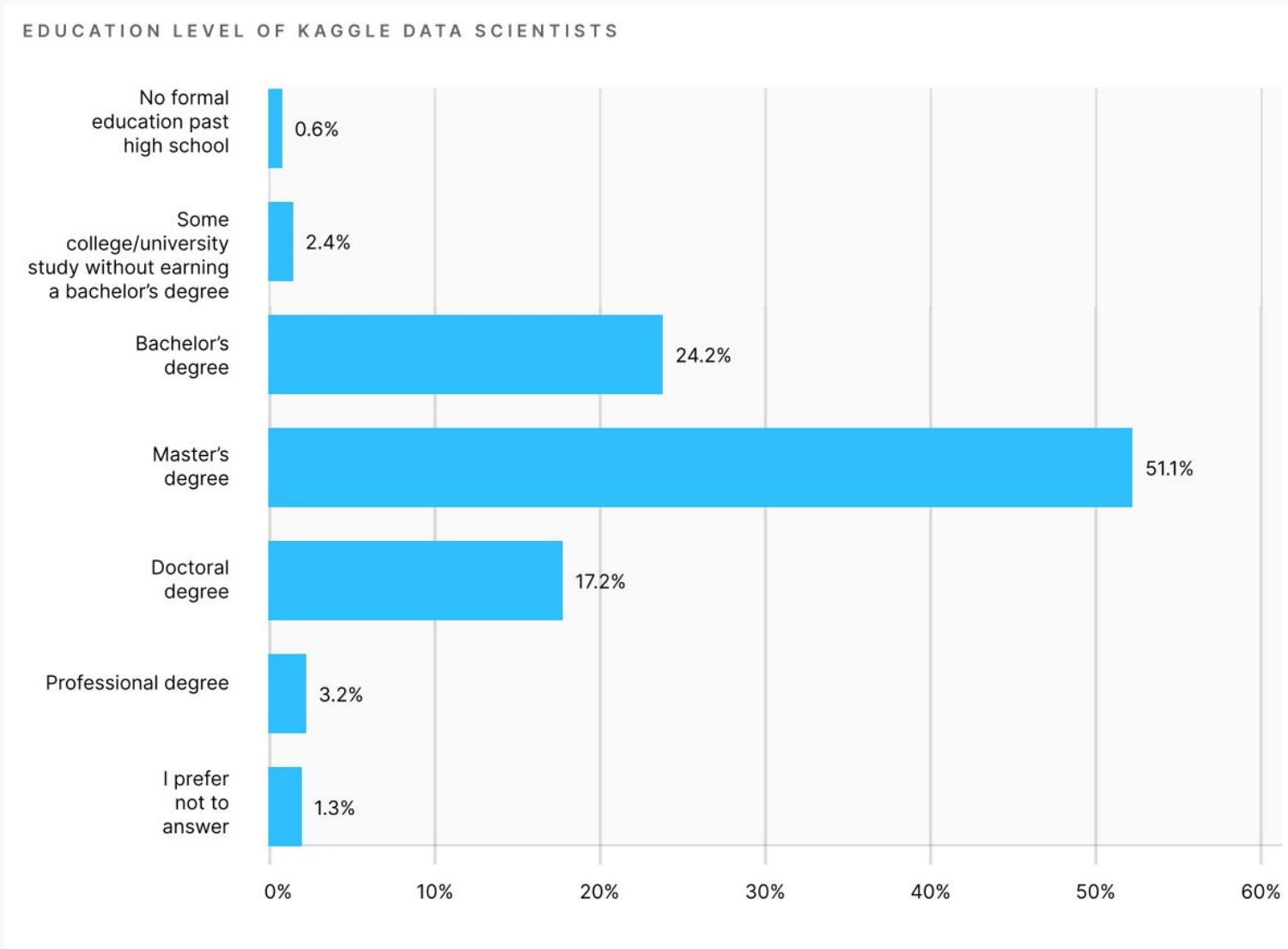
# Key findings: Age



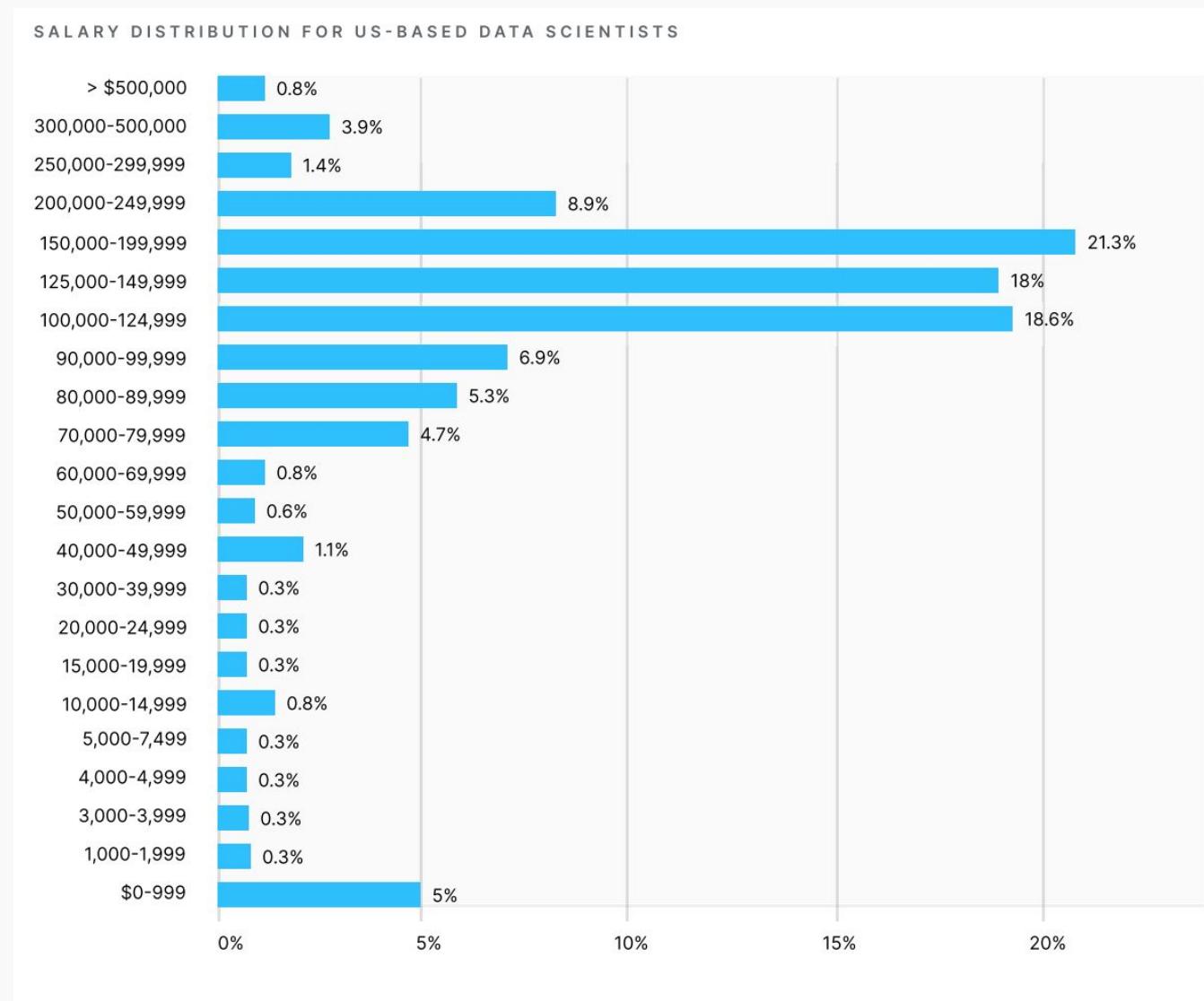
# Key findings: Nationalities



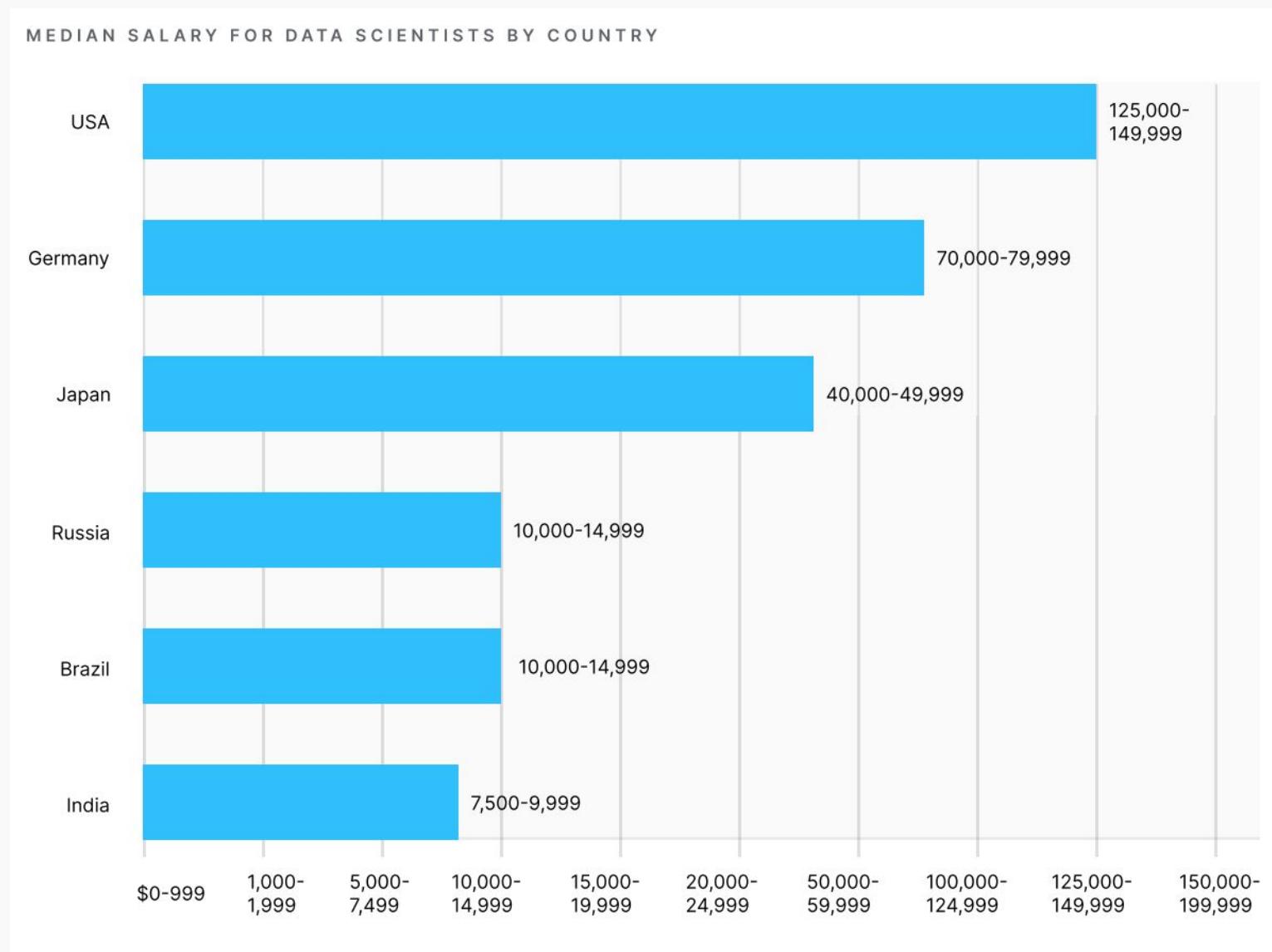
# Key findings: Education



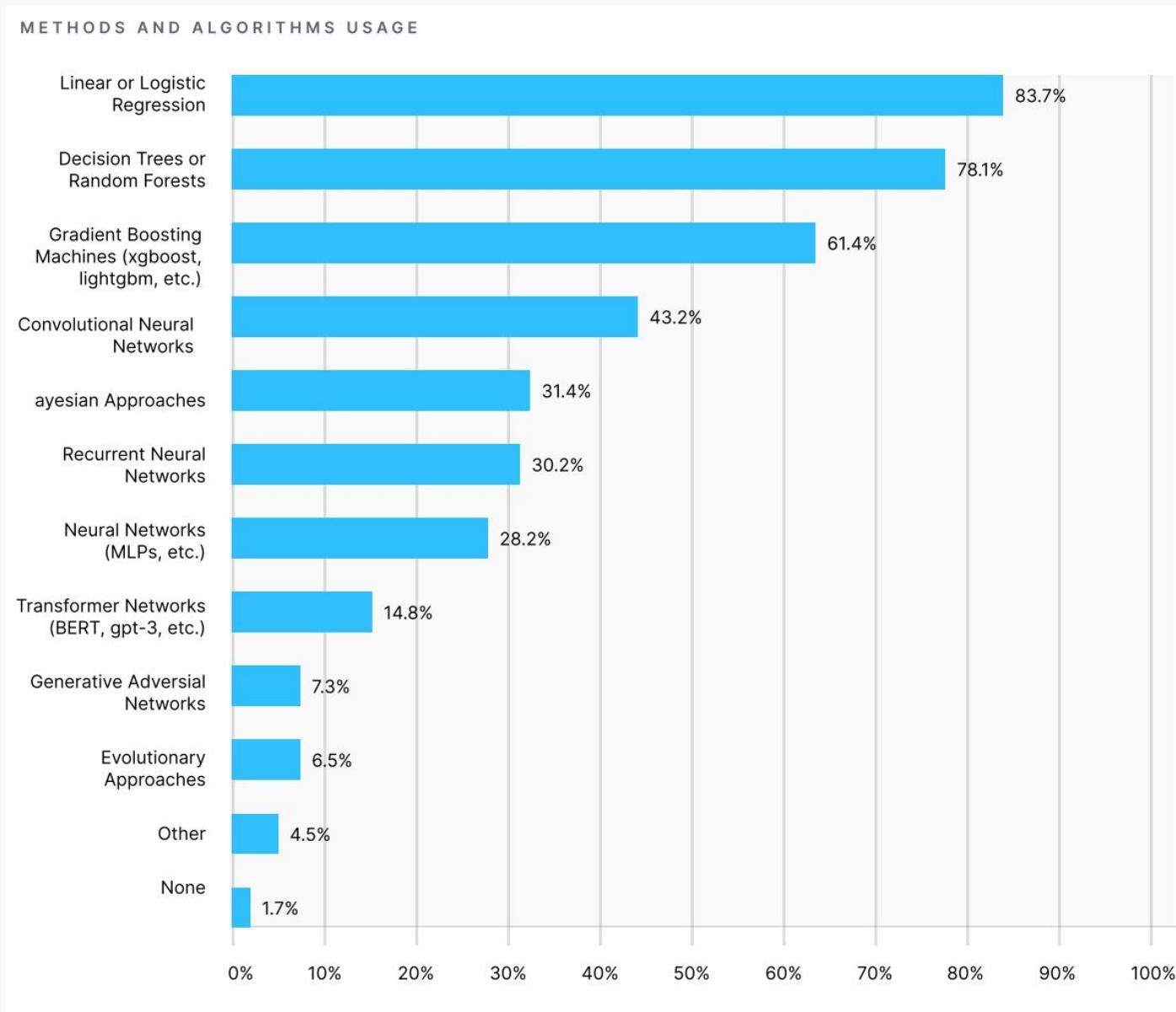
# Key findings: Salary



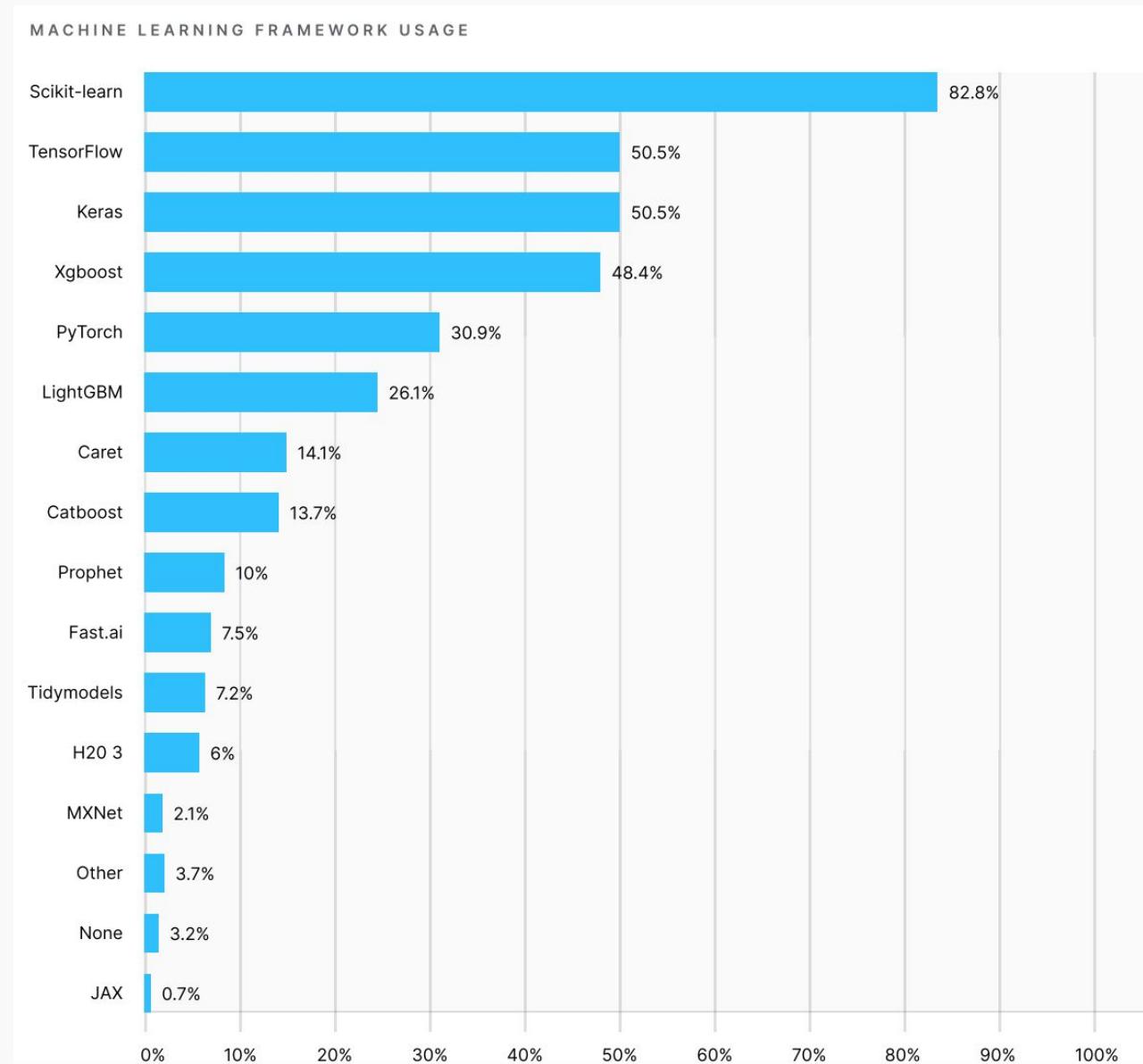
# Key findings: Salary by Country



# Key findings: Methods and Algorithms



# Key findings: ML Frameworks



# Thank You!

