Machine learning from scratch

Lecture 6: Non-linear models, parameter selection

Alexis Zubiolo
alexis.zubiolo@gmail.com

Data Science Team Lead @ Adcash

March 9, 2017

Course outline

Last time, we reviewed the main ideas behind OLS

Course outline

Last time, we reviewed the main ideas behind OLS

This lecture will go a bit further by introducing:

- Regularization
- Non linear models (polynomial kernels)
- Model evaluation
- Parameter selection

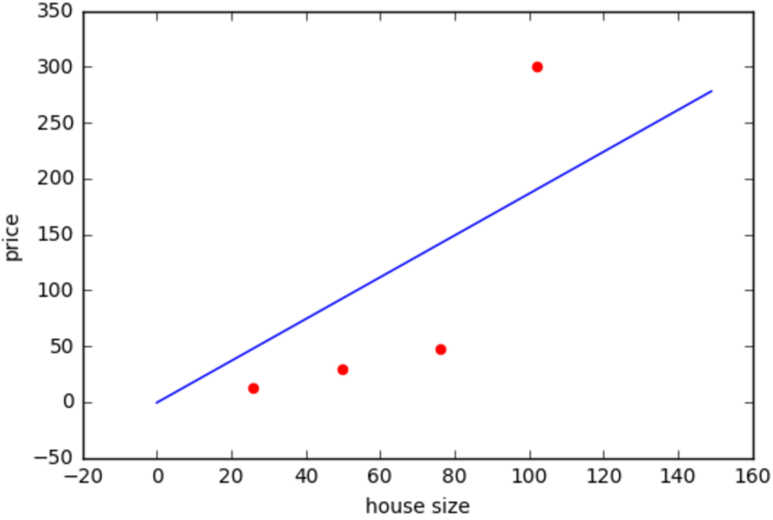
Regularizing models

Outliers and overfitting

Recall from previous course: Outliers can ruin the trained model.

Outliers and overfitting

Recall from previous course: Outliers can ruin the trained model.



Outliers and overfitting

When this happens, we usually notice that one of the weight is big (in this case, θ_1).

Outliers and overfitting

When this happens, we usually notice that one of the weight is big (in this case, θ_1).

How to avoid that?

Outliers and overfitting

When this happens, we usually notice that one of the weight is big (in this case, θ_1).

How to avoid that? By **penalizing big weights** in the model.

Outliers and overfitting

When this happens, we usually notice that one of the weight is big (in this case, θ_1).

How to avoid that? By **penalizing big weights** in the model.

Formally, this consists in rewriting the cost function $J(\theta)$. Initially, we had

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} \ell\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Outliers and overfitting

When this happens, we usually notice that one of the weight is big (in this case, θ_1).

How to avoid that? By **penalizing big weights** in the model.

Formally, this consists in rewriting the cost function $J(\theta)$. Initially, we had

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} \ell \left(\hat{y}^{(i)}, y^{(i)} \right)$$

We can add another term $R(\theta)$ called **regularization**

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} \ell \left(\hat{y}^{(i)}, y^{(i)} \right) + \lambda R(\theta)$$

Outliers and overfitting

When this happens, we usually notice that one of the weight is big (in this case, θ_1).

How to avoid that? By **penalizing big weights** in the model.

Formally, this consists in rewriting the cost function $J(\theta)$. Initially, we had

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} \ell \left(\hat{y}^{(i)}, y^{(i)} \right)$$

We can add another term $R(\theta)$ called **regularization**

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} \ell \left(\hat{y}^{(i)}, y^{(i)} \right) + \lambda R(\theta)$$

where λ is a **hyper-parameter** that quantifies how much we want to penalize big values of θ.

Outliers and overfitting

In the end, $J(\theta)$ is as follows:

$$J(\theta) = L(\theta) + \lambda R(\theta)$$

where

- L is the **loss term**
- R is the **regularization term**
- λ is the **regularization parameter**

Outliers and overfitting

In the end, $J(\theta)$ is as follows:

$$J(\theta) = L(\theta) + \lambda R(\theta)$$

where

- L is the **loss term**
- R is the **regularization term**
- λ is the **regularization parameter**

A commonly used regularization term is often the squared ℓ_2-norm, given by:

$$R(\theta) = \|\theta\|_2^2 = \sum_{j=1}^{d} \theta_j^2$$

ℓ_2-regularized OLS

OLS with this regularization term becomes:

$$J(\theta) = L(\theta) + \lambda R(\theta)$$

ℓ_2-regularized OLS

OLS with this regularization term becomes:

$$J(\theta) = L(\theta) + \lambda R(\theta)$$

This is the new function we have to **minimize**. Once again, we can use a **gradient descent** algorithm. This means we need to **compute the gradient**. By linearity of the gradient operator, we have:

$$\nabla J(\theta) = \nabla L(\theta) + \lambda \nabla R(\theta)$$

ℓ_2-regularized OLS

OLS with this regularization term becomes:

$$J(\theta) = L(\theta) + \lambda R(\theta)$$

This is the new function we have to **minimize**. Once again, we can use a **gradient descent** algorithm. This means we need to **compute the gradient**. By linearity of the gradient operator, we have:

$$\nabla J(\theta) = \nabla L(\theta) + \lambda \nabla R(\theta)$$

We already know $\nabla L(\theta)$, so we just need to find $\nabla R(\theta)$ in order to have $\nabla J(\theta)$.

ℓ_2-regularized OLS

OLS with this regularization term becomes:

$$J(\theta) = L(\theta) + \lambda R(\theta)$$

This is the new function we have to **minimize**. Once again, we can use a **gradient descent** algorithm. This means we need to **compute the gradient**. By linearity of the gradient operator, we have:

$$\nabla J(\theta) = \nabla L(\theta) + \lambda \nabla R(\theta)$$

We already know $\nabla L(\theta)$, so we just need to find $\nabla R(\theta)$ in order to have $\nabla J(\theta)$.

Exercice: Recall

$$R(\theta) = \|\theta\|_2^2 = \sum_{j=1}^{d} \theta_j^2$$

What is $\nabla R(\theta)$?

ℓ_2-regularized OLS

Hint: Recall the definition of the gradient:

$$\nabla R(\theta) = \left[\frac{\partial}{\partial \theta_1} R(\theta), \ldots, \frac{\partial}{\partial \theta_d} R(\theta) \right]$$

ℓ_2-regularized OLS

Hint: Recall the definition of the gradient:

$$\nabla R(\theta) = \left[\frac{\partial}{\partial \theta_1} R(\theta), \ldots, \frac{\partial}{\partial \theta_d} R(\theta)\right]$$

Solution:

$$\begin{aligned}
\frac{\partial}{\partial \theta_k} R(\theta) &= \frac{\partial}{\partial \theta_k} \sum_{j=1}^{d} \theta_j^2 \\
&= \sum_{j=1}^{d} \frac{\partial}{\partial \theta_k} \theta_j^2 \\
&= 2\theta_j
\end{aligned}$$

Conclusion:

$$\nabla R(\theta) = [2\theta_1, \ldots, 2\theta_d]$$

ℓ_2-regularized OLS

Question: Given

$$\nabla R(\theta) = [2\theta_1, \ldots, 2\theta_d]$$

What is gradient descent update rule?

ℓ_2-regularized OLS

Question: Given

$$\nabla R(\theta) = [2\theta_1, \ldots, 2\theta_d]$$

What is gradient descent update rule?

Hint: Recall the gradient descent update rule:

$$\theta_k := \theta_k - \alpha \frac{\partial}{\partial \theta_k} J(\theta)$$

Solution: We have

$$\frac{\partial}{\partial \theta_k} L(\theta) = (y - h(\boldsymbol{x}))\, x_k \qquad \text{and} \qquad \frac{\partial}{\partial \theta_k} R(\theta) = 2\theta_k$$

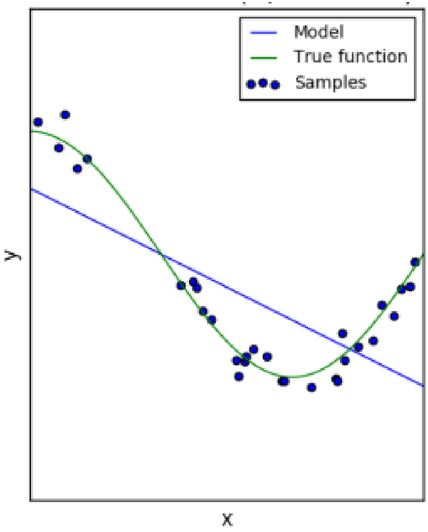
so

$$\theta_k := \theta_k - \alpha((y - h(\boldsymbol{x}))\, x_k - 2\lambda\theta_k)$$

More sophisticated models

Non-linearity

We might want to apply OLS to this data:



Non-linearity

We'd rather try to fit **a polynomial** on the data:

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_d x^d = \sum_{k=0}^{d} \theta_d x^d$$

Non-linearity

We'd rather try to fit **a polynomial** on the data:

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_d x^d = \sum_{k=0}^{d} \theta_d x^d$$

Se need to find $\theta_0, \theta_1, \ldots, \theta_d$, which can be achieved by defining a new feature vector $\phi(x) = [1, x, x^2, \ldots, x^d]$ and define the prediction model

$$\hat{y} = \theta^T h(\phi(x)) = \sum_{k=0}^{d} \theta_d x^d$$

Non-linearity

We'd rather try to fit **a polynomial** on the data:

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_d x^d = \sum_{k=0}^{d} \theta_d x^d$$

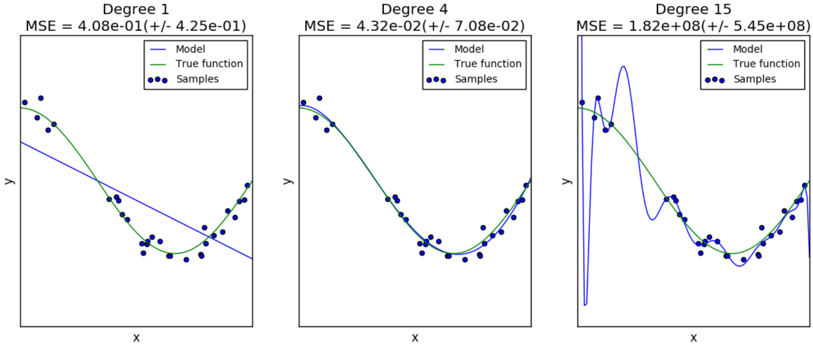
Se need to find $\theta_0, \theta_1, \ldots, \theta_d$, which can be achieved by defining a new feature vector $\phi(x) = [1, x, x^2, \ldots, x^d]$ and define the prediction model

$$\hat{y} = \theta^T h(\phi(x)) = \sum_{k=0}^{d} \theta_d x^d$$

OLS can train θ and we have a **polynomial model**. The trick is to apply a **non-linearity mapping** ϕ.

Non-linearity

Problem: We would need to find an appropriate value for the degree d, because:



Hyperparameters

This is a **more general problem**: **Tuning hyperparameters**. The current version of OLS we have has several parameters:

- The regularization parameter λ
- The kernel degree d
- The learning rate α

Hyperparameters

This is a **more general problem**: **Tuning hyperparameters**. The current version of OLS we have has several parameters:

- The regularization parameter λ
- The kernel degree d
- The learning rate α

As we've seen, all these 3 parameters can have a **dramatic impact** on the quality of our prediction model! Hence, we need to **tune them** properly.

Model evaluation
Parameter selection

Train-test split

As we saw with OLS, ML algorithms usually rely on **many parameters**. How to **tune** them properly given a data set?

Train-test split

As we saw with OLS, ML algorithms usually rely on **many parameters**. How to **tune** them properly given a data set?

The most commonly used principle is the **train-test split**:

- **Split the data** into a training set and a test set
- **Train** on the training set
- **Test** on the test set

Train-test split

As we saw with OLS, ML algorithms usually rely on **many parameters**. How to **tune** them properly given a data set?

The most commonly used principle is the **train-test split**:

- **Split the data** into a training set and a test set
- **Train** on the training set
- **Test** on the test set

This is often referred to as **cross-validation**.

Cross-validation

Standard technique: Hold-out cross-validation:

- Train on a part of the data (e.g. 70%)
- Test on the remaining data (e.g. 30%)

Cross-validation

Standard technique: Hold-out cross-validation:

- Train on a part of the data (e.g. 70%)
- Test on the remaining data (e.g. 30%)

Another standard technique: k-**fold cross-validation**

- Split the data into k (equally-sized) folds
- Remove 1 fold (= test fold)
- Train on the other folds
- Test on the removed fold
- Do it for all the folds

Small data set

Suppose you have **a small data set**. How to evaluate a classifier in this case?

Small data set

Suppose you have **a small data set**. How to evaluate a classifier in this case?

k-fold cross-validation? Even with $k = 2$, it would make the training set even smaller and make it hard to fit a proper model.

Small data set

Suppose you have **a small data set**. How to evaluate a classifier in this case?

k-fold cross-validation? Even with $k = 2$, it would make the training set even smaller and make it hard to fit a proper model.

Other option: **Leave-one-out** (LOO) cross-validation:

Small data set

Suppose you have **a small data set**. How to evaluate a classifier in this case?

k-fold cross-validation? Even with $k = 2$, it would make the training set even smaller and make it hard to fit a proper model.

Other option: **Leave-one-out** (LOO) cross-validation:

- **Remove 1 sample** from the data set
- Train on **all the other samples**
- Test on the sample you've removed
- **Evaluate** the prediction
- Do it **for each sample of the data set**
- **Aggregate** the evaluations

Small data set

Suppose you have **a small data set**. How to evaluate a classifier in this case?

k-fold cross-validation? Even with $k = 2$, it would make the training set even smaller and make it hard to fit a proper model.

Other option: **Leave-one-out** (LOO) cross-validation:

- **Remove 1 sample** from the data set
- Train on **all the other samples**
- Test on the sample you've removed
- **Evaluate** the prediction
- Do it **for each sample of the data set**
- **Aggregate** the evaluations

Remark: This could lead to many iterations even if the data set is small.

Small data set

Suppose you have **a small data set**. How to evaluate a classifier in this case?

k-fold cross-validation? Even with $k = 2$, it would make the training set even smaller and make it hard to fit a proper model.

Other option: **Leave-one-out** (LOO) cross-validation:

- **Remove 1 sample** from the data set
- Train on **all the other samples**
- Test on the sample you've removed
- **Evaluate** the prediction
- Do it **for each sample of the data set**
- **Aggregate** the evaluations

Remark: This could lead to many iterations even if the data set is small.

Alternative: Leave-p-out (LPO). LOO is LPO with $p = 1$.

Parameter selection

One of the goals of model evaluation is to select a **good model**.

Parameter selection

One of the goals of model evaluation is to select a **good model**.

Hence we want to choose one (or several) parameters. How do we do?

Parameter selection

One of the goals of model evaluation is to select a **good model**.

Hence we want to choose one (or several) parameters. How do we do?

Most classic way: a **grid-search**

- For each parameter, define a set of possible values
- For each parameter combination, train/test the model
- Pick the parameter combination which gives best results

Parameter selection

One of the goals of model evaluation is to select a **good model**.

Hence we want to choose one (or several) parameters. How do we do?

Most classic way: a **grid-search**

- For each parameter, define a set of possible values
- For each parameter combination, train/test the model
- Pick the parameter combination which gives best results

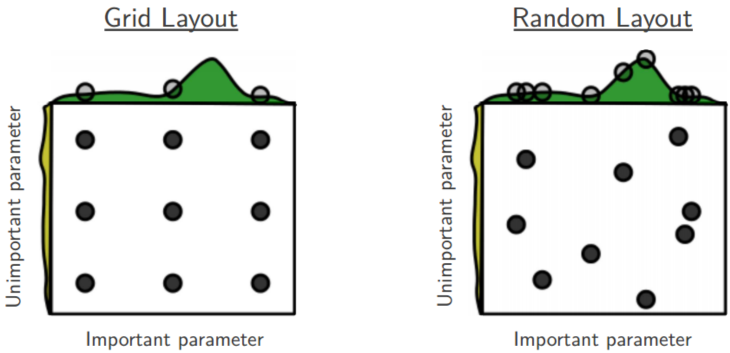
Important note: Hyper-parameter ranges vary a lot from an application to another. It is **data-dependent**.

Grid search vs random search

Random search is a more and more popular alternative to grid search, especially in this case:

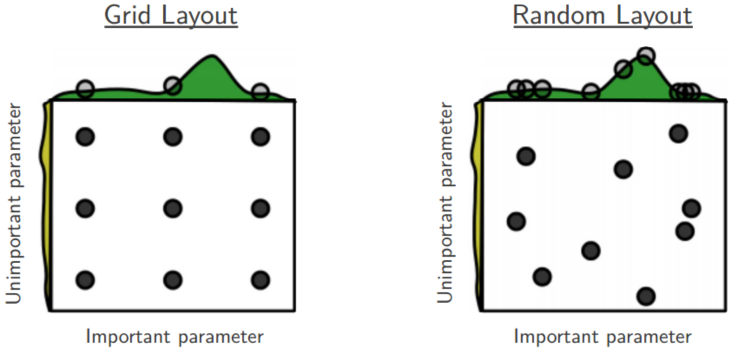
Grid search vs random search

Random search is a more and more popular alternative to grid search, especially in this case:



Grid search vs random search

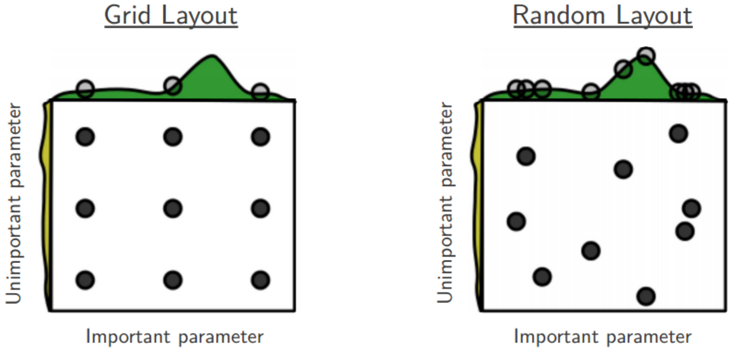
Random search is a more and more popular alternative to grid search, especially in this case:



In any case, you need to guess upper/lower bounds on the parameters.

Grid search vs random search

Random search is a more and more popular alternative to grid search, especially in this case:



In any case, you need to guess upper/lower bounds on the parameters.

Practical note: Each parameter combination can be trained/tested separately => possibility to distribute the tasks

Conclusion

In this lecture, we've seen:

- How to avoid overfitting by regularizing the model
- How to evaluate models
- How to tune parameters automatically

Conclusion

In this lecture, we've seen:

- How to avoid overfitting by regularizing the model
- How to evaluate models
- How to tune parameters automatically

During the next lecture, we will work on implementing regularization to the OLS algorithm and cross-validating it and switch to classification if the time allows it.

Thank you! Questions?