# Regression Algorithms

For practical machine learning, it's typically better to **allow more complexity in the models, and then implement safeguards against overfitting**.

* The reason is that if the models are too simple, they can't learn complex patterns no matter how much data you collect.
* However, if the models are more complex, we can use tactics to avoid overfitting. In addition, collecting more data will naturally reduce the chance of overfitting.
* Regularization, ensemble, train/test splitting, cross-validation, etc. prevent overfitting.

## Flaws of Linear Regression

In practice, simple linear regression models are rarely optimal. Their main advantage is that they are easy to interpret and understand.

Simple linear regression suffers from two major flaws:

* It's prone to overfitting with many input features.
* It cannot easily express non-linear relationships.

We already saw how Decision Trees can help with the second flaw, but let's take a closer at the first flaw.

***FLAW 1:*** *It's prone to overfitting with many input features.*

Let's take an example with simple linear regression:

* Let's say we have **100 observations** in our training dataset.
* Now, let's say we also have **100 input features**.
* Assuming most of those input features are completely worthless (i.e. they are **pure noise**).
* Even so, if we fit a linear regression model with those 100 features, we can still perfectly **"memorize" the training set**.
* Since we have 100 input features and 100 training observations, **we have enough coefficient combinations to memorize every individual observation.**
* we would have perfect accuracy on the training data. However, **this model would be over fit,** and would perform poorly on unseen data because it hasn't learned the true underlying patterns; it has only memorized the training data itself.

As you can see, we need a way to safeguard against this! Enter: **Regularization**.

## 1. Regularized Regression

Next, we'll discuss the first technique we'll use to improve the performance of our models**.**

***Regularization*** (in the context of linear regression) is a technique used to prevent overfitting by artificially penalizing model coefficients.

* It does so by adding a **penalty factor** to the cost function.
* This dampens coefficients or removes features entirely, depending on the type of penalty used.
* The "strength" of that penalty factor is **tunable.**

There are two types of penalties:

* **L1** penalizes the absolute size of model coefficients.
* **L2** penalizes the squared size of model coefficients.

There are 3 regularized regression algorithms:

* Lasso
* Ridge
* Elastic-Net

## I. LASSO Regression

They correspond with the amount of L1 and L2 penalty included in the cost function.

* Lasso Regression relies on the **L1** penalty.
* Lasso, or LASSO, stands for **L**east **A**bsolute **S**hrinkage and **S**election **O**perator.
* Lasso regression completely relies on the **L1** penalty (**absolute size**).
* Practically, this leads to coefficients that can be exactly 0 (**automatic feature selection**).
* Remember, **λ** (the "strength" of the penalty) should be **tuned**. A stronger penalty leads to more coefficients pushed to zero.

Scikit-Learn comes with a **Lasso** class that you can import from **sklearn.linear\_model**, like so:

*# Import Lasso Regression*

from sklearn.linear\_model **import** Lasso

## II. Ridge Regression

Ridge Regression relies on the **L2** penalty .

* Ridge stands for just ridge.
* Ridge regression completely relies on the **L2** penalty (**squared**).
* Practically, this leads to smaller coefficients, but it doesn't force them to 0 (**feature shrinkage**).
* Remember, **λ** (the "strength" of the penalty) should be **tuned**. A stronger penalty leads coefficients pushed closer to zero.

Scikit-Learn comes with a **Ridge** class that you can import from **sklearn.linear\_model**, like so:

*# Import Ridge Regression*

from sklearn.linear\_model **import** Ridge

## III. Enet Regression

Elastic-Net is a compromise between Lasso and Ridge.

Elastic-Net combines both **L1** and **L2** penalties.

* The ratio of **L1** and **L2** penalties should be **tuned**.
* The strength of the penalties should also be **tuned**.
* Ridge and Lasso regression are **special cases** of Elastic-Nets if you set the ratio to completely favor one of the penalty types over the other.

Scikit-Learn comes with a **ElasticNet** class that you can import from **sklearn.linear\_model**, like so:

*# Import Elastic Net*

from sklearn.linear\_model **import** ElasticNet

*Why we bothered importing classes for Ridge and Lasso if Elastic-Net combines them?*

* It's because Scikit-Learn's implementation of ElasticNet can act weird if we set the penalty ratio to 0 or 1. Therefore, we should use the separate Ridge and Lasso classes for those special cases.
* Also, for some problems, Lasso or Ridge are just as effective while being easier to tune.

There is a **penalty strength** that must be tuned. And in Scikit-Learn, the argument for tuning that penalty strength is alpha

For example, we can reduce the penalty strength by setting alpha=0.1.

* The default value is alpha=1.0
* This should reduce the regularization amount and end up with fewer coefficients forced to 0.

## 2. Ensemble Methods

Linear Regression suffers from **two** main flaws:

***FLAW 2:*** *It cannot easily express non-linear relationships.*

Decision trees are used model non-linear relationships. However, unconstrained decision trees are also prone to overfitting because they can keep growing until each leaf memorizes an individual training observation.

## I. Random Forest

Random forests train a large number of "strong" decision trees and combine their predictions through bagging.

* **Bagging** attempts to reduce the chance of overfitting complex models. It trains a large number of "strong" learners in parallel. A strong learner is a model that's allowed to have high complexity. It then combines all the strong learners to "smooth out" their predictions.

In addition, there are two sources of "randomness" for random forests:

* Each decision tree is only allowed to choose from a random subset of features to split on.
* Each decision tree is only trained on a random subset of observations (a process called **resampling**).

Scikit-Learn comes with a **RandomForestRegressor** class that you can import from **sklearn.ensemble**, like so:

*# Import Random Forest*

from sklearn.ensemble **import** RandomForestRegressor

## I. Boosted Trees

Boosted trees train a sequence of "weak", *constrained* decision trees and combine their predictions through boosting.

* **Boosting** attempts to improve the predictive flexibility of simple models. It trains a large number of "weak" learners in sequence. A weak learner is a model that has limited complexity. Each one in the sequence focuses on learning from the mistakes of the one before it. It then combines all the weak learners into a single strong learner.

In addition, the component trees have some special properties:

* Each decision tree is allowed a **maximum depth**, which should be tuned.
* Each decision tree in the sequence tries to correct the prediction errors of the one before it.

Scikit-Learn comes with a **GradientBoostingRegressor** class that you can import from **sklearn.ensemble**, like so:

*# Import Gradient Boosted Trees*

from sklearn.ensemble **import** GradientBoostingRegressor

In practice, boosted trees tend to have the high performance ceilings.

* They often beat many other types of models after proper tuning.
* They are more complicated to tune than random forests.

While bagging and boosting are both ensemble methods, they approach the problem from opposite directions. Bagging uses complex base models and tries to "smooth out" their predictions, while boosting uses simple base models and tries to "boost" their aggregate complexity.