Model Tuning

Recap from Under the Hood

Problem setting

- X
  = features
- y
  = target =
  $h(X, \beta) + error$
- h
  = hypothesis function (Linear, Logistic Regression, etc.)

Parameters of the model:
β

- computed automatically during `.fit()`
- by minimizing
  $L(\beta)$

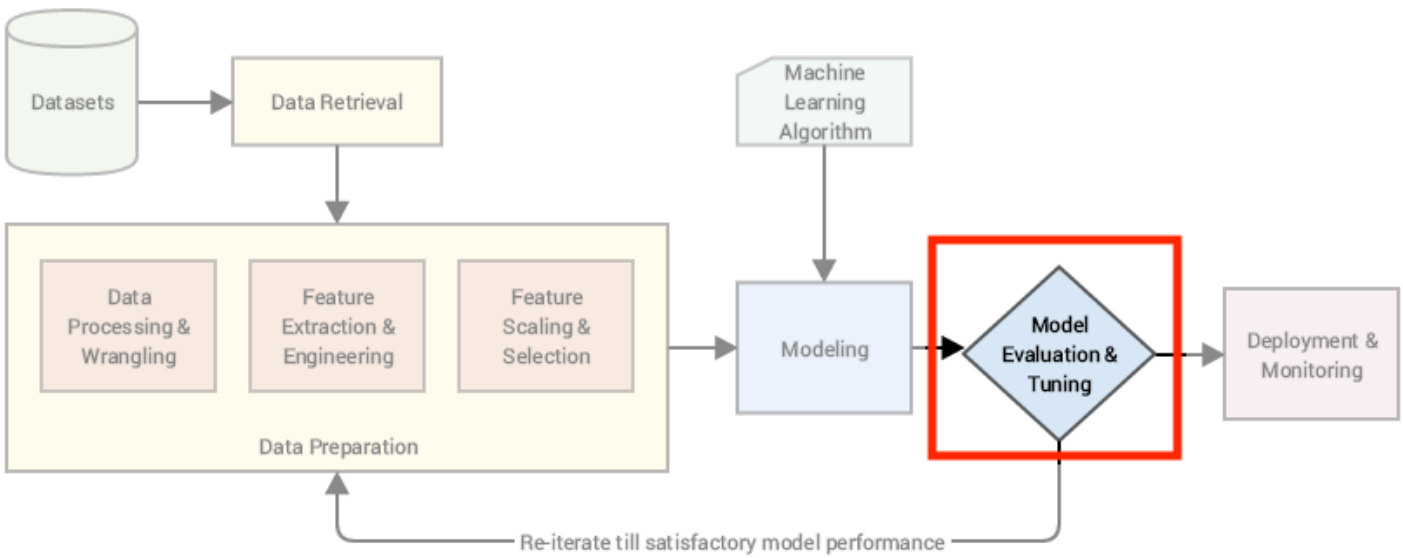
Hyperparameters of the model (chosen manually)

- Loss Function
  L
  (MSE, Log-Loss, etc.)
- loss parameters (learning_rate, eta0, etc.)
- solver = method used to minimize
  L
  ('newton', 'sgd', etc.)
- model specificities ('n_neighbors', 'kernel', etc.)
- and more

Plan

1. Model Complexity
2. Regularization
3. Model Tuning: Grid Search & Random Search
4. Support Vector Machines (Margin Classifiers)
5. Kernel Tricks

The Tuning Stage

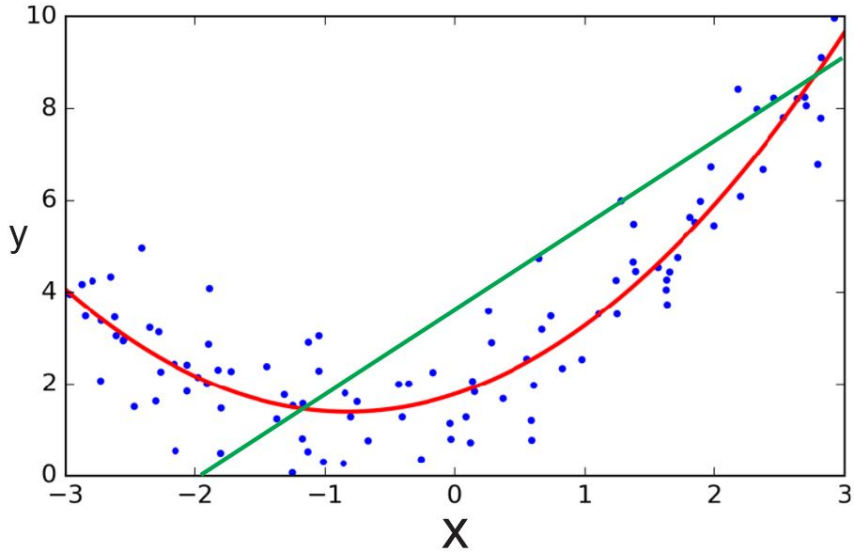


1. Model Complexity

Let's remember Linear Regression:

$$Y = \beta_0 + \beta_1 X_1 + \epsilon$$

❓ What about non-linear behavior as below?



👉 If we add a new transformed feature
X_1^2
we have a better fit (in red)

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_1^2 + \ldots + \epsilon$$

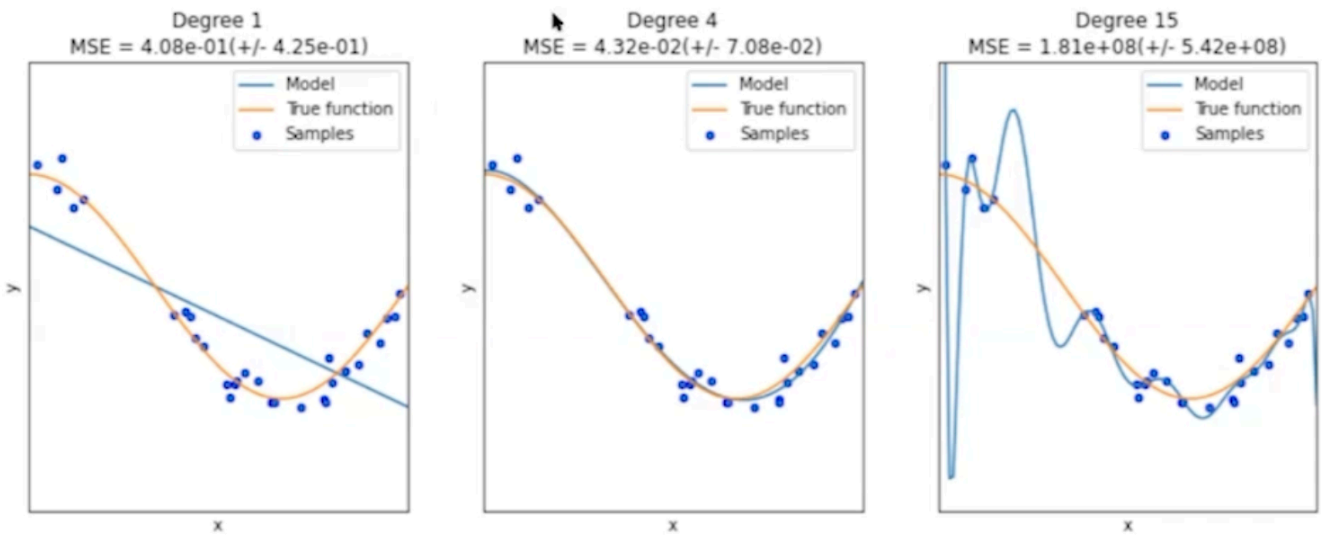
🤔 We could engineer very complex features:
$X^3, X^6, (X_1^2 * X_2^5), \ldots$

Our
R^2
will keep increasing with every additional feature!

> We just solved Data Science!

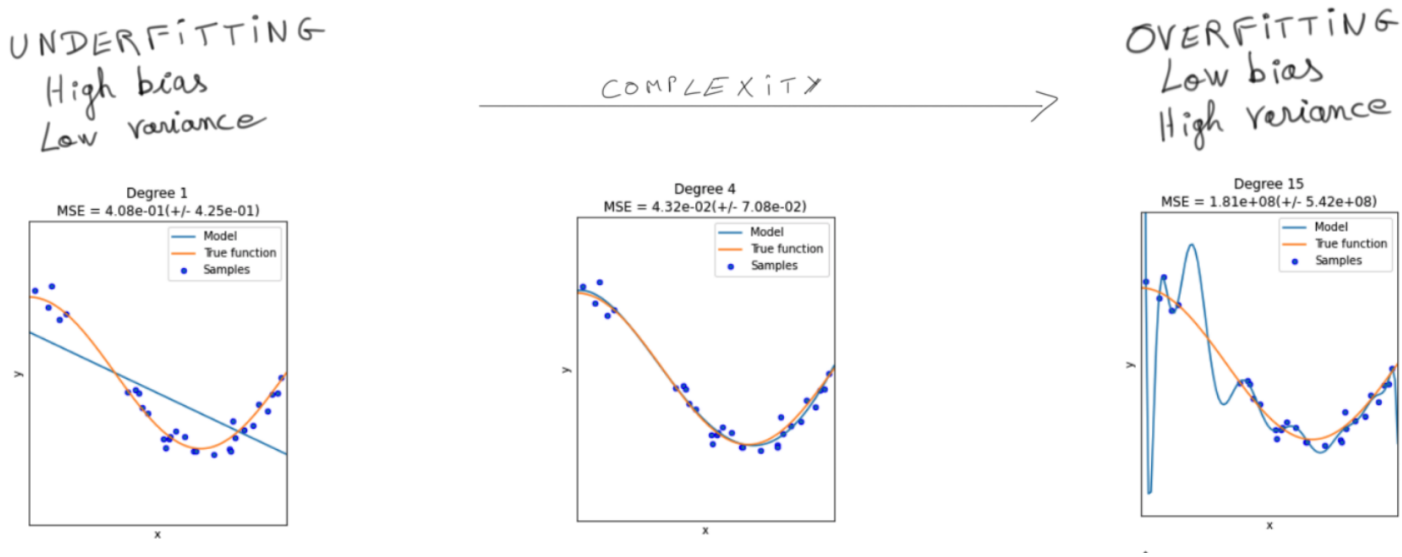
In [ ]:
```
def train_best_model_ever(X,y):
    while True:
        model = LinearRegression()
        model.fit(X, y)
        if calculate_r2(model, X, y) < 0.999999:
            X = add_more_crazy_features(X)
        else:
            return model
```

Overfitting and Underfitting



What's wrong with the first model? With the third?

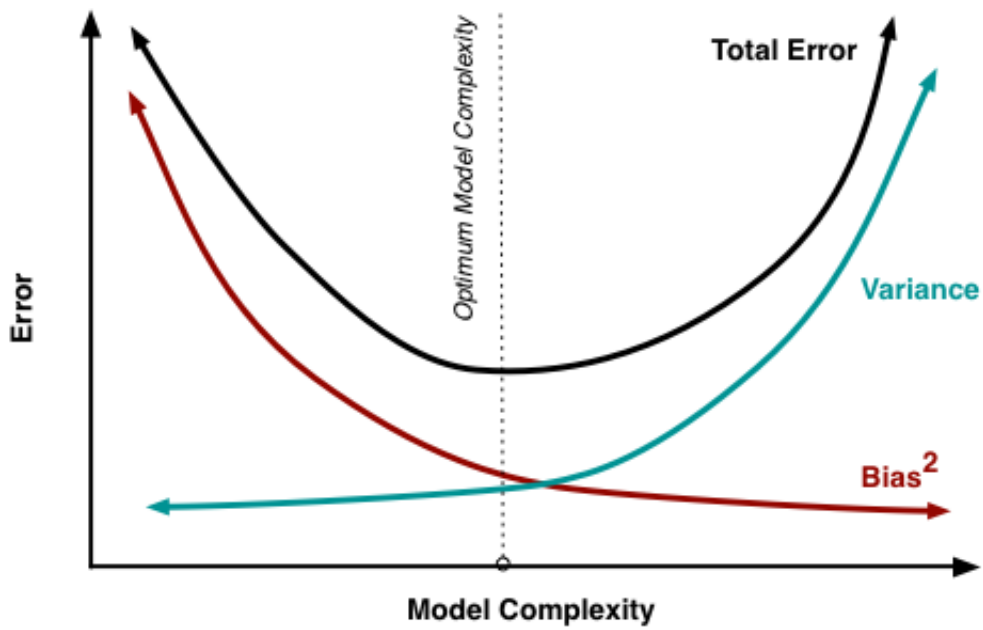
- The first one does not capture all the information in the data (**high bias** model)
- The third one finds signals that aren't there (noise); it will not generalize well to new data points (**high variance** model)



The Bias-Variance Tradeoff

One of the most important concepts in Data Science!

Look at what happens in reality when measuring the error on an unseen **test set**:

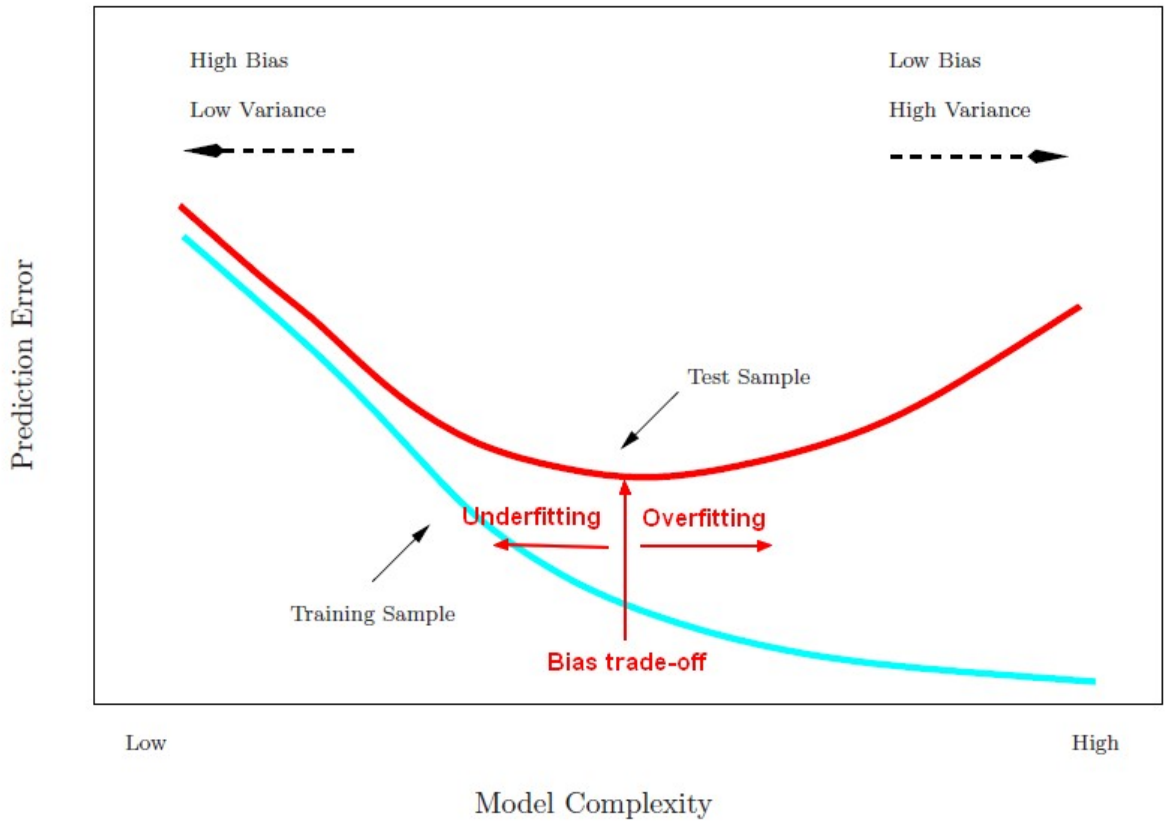


$$TotalError = Bias^2 + Variance + IrreducibleError$$

📚 [Great read (http://scott.fortmann-roe.com/docs/BiasVariance.html)](http://scott.fortmann-roe.com/docs/BiasVariance.html)

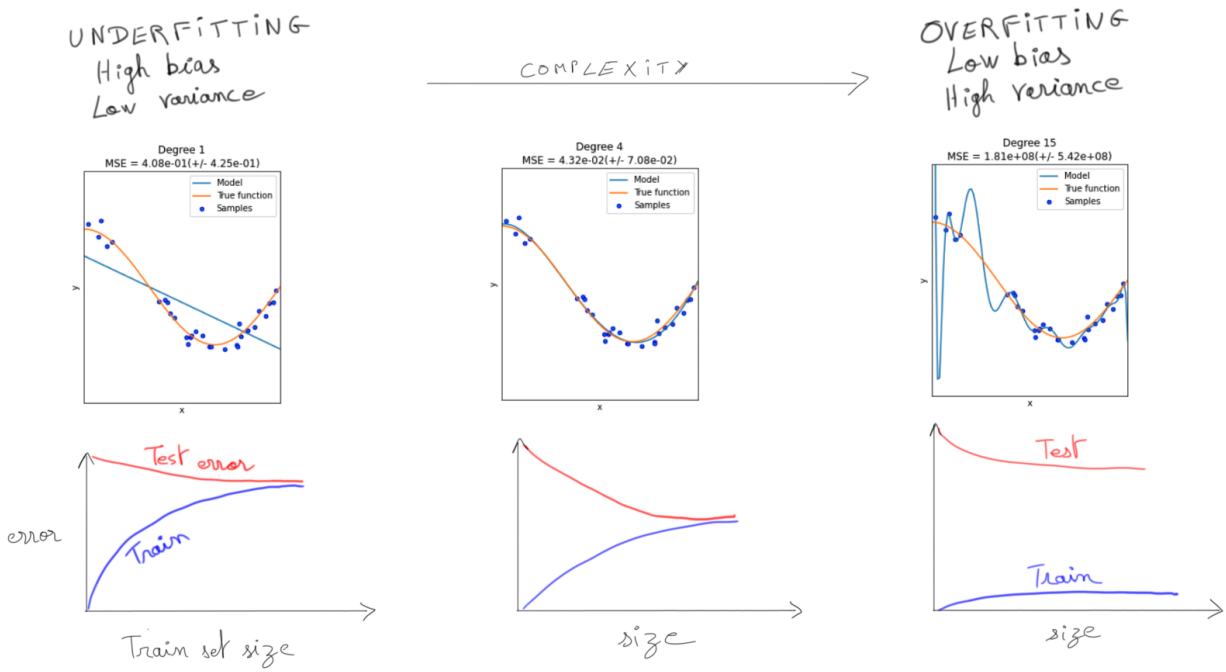
❓ How do we find the optimal model complexity?

The one that minimizes the **test error** on an unseen dataset



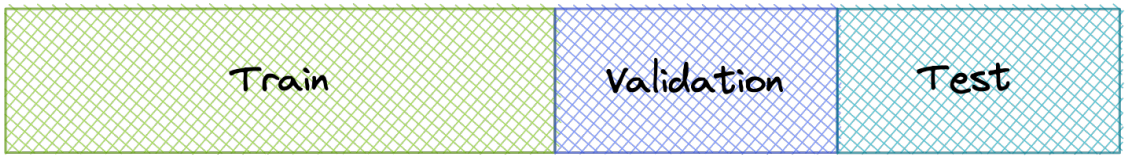
🤔 Don't have time to try many models?

Use **learning curves** to at least diagnose the only one you have!



❗ **Data Leakage Reminder** ❗

Always diagnose your model using a **validation set**!



👉 **Cross-validate** instead of using a single holdout validation set to generalize better!

Solutions for Overfitting

> *Simplify* your model *relative* to your data

- Get more observations
- Feature selection (manual or automated (https://scikit-learn.org/stable/modules/feature_selection.html))
- Dimensionality reduction (Unsupervised Learning)
- Early stopping (Deep Learning)
- **Regularization** of your Loss function

2. Regularization

Regularization means adding a **penalty term** to the Loss that **increases** with β

$$RegularizedLoss = Loss(X, y, \beta) \ + Penalty(\beta)$$

👉 Penalizes large values for β_i

👉 Forces model to shrink certain coefficients or even select less features
👉 Prevents overfitting

$\hat{y}$

The two most famous Regularization penalties are:

$$Ridge(X, y, \beta) = \underbrace{\sum_{i=0}^{n-1}(y_i - (\beta_0 + \beta_1 x_1 + \ldots \beta_p x_p))^2}_{Loss(X,y,\beta)} + \alpha \sum_{j=1}^{p} {\beta_j}^2$$

$$Lasso(X, y, \beta) = \underbrace{\sum_{i=0}^{n-1}(y_i - (\beta_0 + \beta_1 x_1 + \ldots \beta_p x_p))^2}_{Loss(X,y,\beta)} + \alpha \sum_{j=1}^{p} |$$

❗ ***Warning*** ❗

When regularizing a Regression with a L2 penalty or with a L1 penalty:

- We regularize
  $\beta_1, \ldots, \beta_p$
- We do NOT regularize the intercept
  β_0
  - Keep in mind that Ridge reduces the influence of features with a non-significant coefficient
  - Keep in mind that Lasso shrinks them to zero
  - But
    β_0
    is not associated with any feature!

New hyper-parameter
α

- Dictates **how much** the model is **regularized**
- Large
  α
  values force model complexity to decrease 👉 ⤵ variance, ⤴ bias

The sum starts from j = 1, we do **not** penalize the intercept coefficient

🖥️ Comparing Ridge vs. Lasso?

In [ ]:
```
X, y = datasets.load_diabetes(return_X_y=True, as_frame=True)
X.head()
```

Out[ ]:

	age	sex	bmi	bp	s1	s2	s3	s4	
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.01
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.06
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	-0.002592	0.00
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.02
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.03

In [ ]:
```
y.head()
```

Out[ ]:
```
0    151.0
1     75.0
2    141.0
3    206.0
4    135.0
Name: target, dtype: float64
```

⚠️ Always **scale** your features before regularization to penalize each
β_i
fairly

(already scaled here)

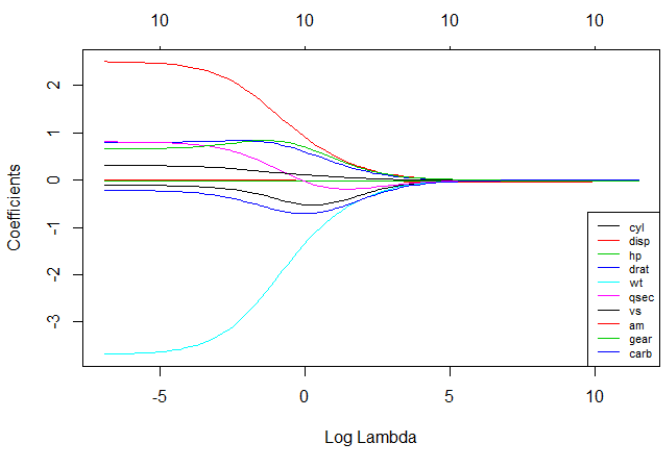
In [ ]:
```
from sklearn.linear_model import Ridge, Lasso, LinearRegression

linreg = LinearRegression().fit(X, y)
ridge = Ridge(alpha=0.2).fit(X, y)
lasso = Lasso(alpha=0.2).fit(X, y)

coefs = pd.DataFrame({
    "coef_linreg": pd.Series(linreg.coef_, index = X.columns),
    "coef_ridge": pd.Series(ridge.coef_, index = X.columns),
    "coef_lasso": pd.Series(lasso.coef_, index= X.columns)})\

coefs\
    .map(lambda x: int(x))\
    .style.map(lambda x: 'color: red' if x == 0 else 'color: black')
```

Out[ ]:

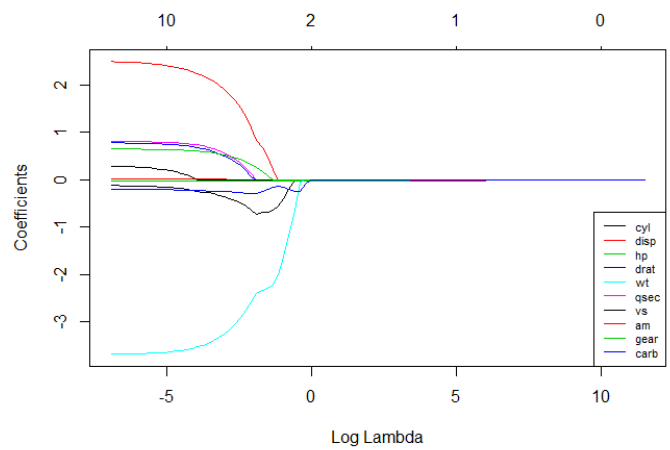
	coef_linreg	coef_ridge	coef_lasso
age	-10	7	0
sex	-239	-182	-75
bmi	519	457	511
bp	324	284	234
s1	-792	-48	0
s2	476	-78	0
s3	101	-189	-170
s4	177	119	0
s5	751	400	450
s6	67	97	0

What happens when alpha increases?

Ridge



Lasso



- Increasing α in Ridge will only shrink parameters **toward 0**
- Increasing α in Lasso can shrink parameters **to 0** (natural feature selector)

ElasticNet = Lasso & Ridge Weighted Average

$L = \|y - \hat{y}$

2 hyper-parameters to fine-tune (
α
,
λ
)

```
from sklearn.linear_model import ElasticNet
model = ElasticNet(alpha=1, l1_ratio=0.2)
```

🤔 Which features are penalized?

In [ ]:
```
# Let's check the p-values of our features before regularization
import statsmodels.api as sm

ols = sm.OLS(y, sm.add_constant(X)).fit()
ols.summary()
```

Out[ ]:

OLS Regression Results

Dep. Variable:	target	**R-squared:**	0.518
Model:	OLS	**Adj. R-squared:**	0.507
Method:	Least Squares	**F-statistic:**	46.27
Date:	Thu, 09 Feb 2023	**Prob (F-statistic):**	3.83e-62
Time:	23:59:47	**Log-Likelihood:**	-2386.0
No. Observations:	442	**AIC:**	4794.
Df Residuals:	431	**BIC:**	4839.
Df Model:	10		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	152.1335	2.576	59.061	0.000	147.071	157.196
age	-10.0099	59.749	-0.168	0.867	-127.446	107.426
sex	-239.8156	61.222	-3.917	0.000	-360.147	-119.484
bmi	519.8459	66.533	7.813	0.000	389.076	650.616
bp	324.3846	65.422	4.958	0.000	195.799	452.970
s1	-792.1756	416.680	-1.901	0.058	-1611.153	26.802
s2	476.7390	339.030	1.406	0.160	-189.620	1143.098
s3	101.0433	212.531	0.475	0.635	-316.684	518.770
s4	177.0632	161.476	1.097	0.273	-140.315	494.441
s5	751.2737	171.900	4.370	0.000	413.407	1089.140
s6	67.6267	65.984	1.025	0.306	-62.064	197.318

Omnibus:	1.506	**Durbin-Watson:**	2.029
Prob(Omnibus):	0.471	**Jarque-Bera (JB):**	1.404
Skew:	0.017	**Prob(JB):**	0.496
Kurtosis:	2.726	**Cond. No.**	227.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [ ]:
```
coefs_with_p_value
```

Out[ ]:

	coef_linreg	coef_ridge	coef_lasso	p-values (%)
age	-10.009866	7.728551	0.000000	86
sex	-239.815644	-182.946743	-75.612133	0
bmi	519.845920	457.176049	511.404133	0
bp	324.384646	284.516603	234.508645	0
s1	-792.175639	-48.471100	-0.000000	5
s2	476.739021	-78.867888	-0.000000	16
s3	101.043268	-189.672329	-170.214828	63
s4	177.063238	119.682742	0.000000	27
s5	751.273700	400.706510	450.678492	0
s6	67.626692	97.378604	0.224852	30

☝️ Regularization *tends to* **penalize** features that are **not statistically significant**

Conclusions

👉 **Regularize** when you think you are **overfitting** (e.g. Learning Curves not converging)

👉 **Ridge** when you believe all coefficients may have an impact

👉 **Lasso** as a feature selection tool (much better for interpretability!)

✅ Regularization is almost always appropriate

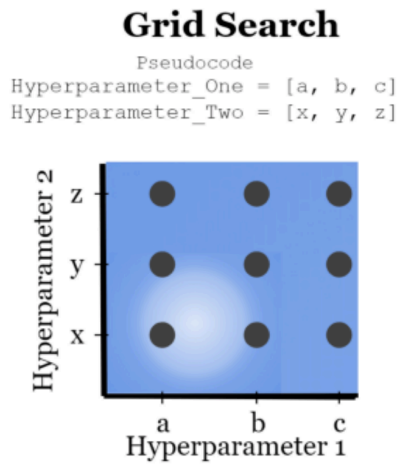
- Ridge is often turned on by default in most Machine Learning models
- You just have to tune the regularization parameter

3. Model Tuning

❓ How to choose the best hyper-parameters (e.g: alpha)

The Grid Search Method

Explores different hyperparameter value combinations to find the combination which optimizes performance



1. Hold out a *validation set* (never use the test set for model tuning!)
2. Select which grid of values of hyper-parameters to try out
3. For each combination of values, measure your performance on the *validation set*
4. Select hyper-parameters that produce the best performance

💻 Let's (manually) fine-tune a linear model with [ElasticNet (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html)](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html) regularization

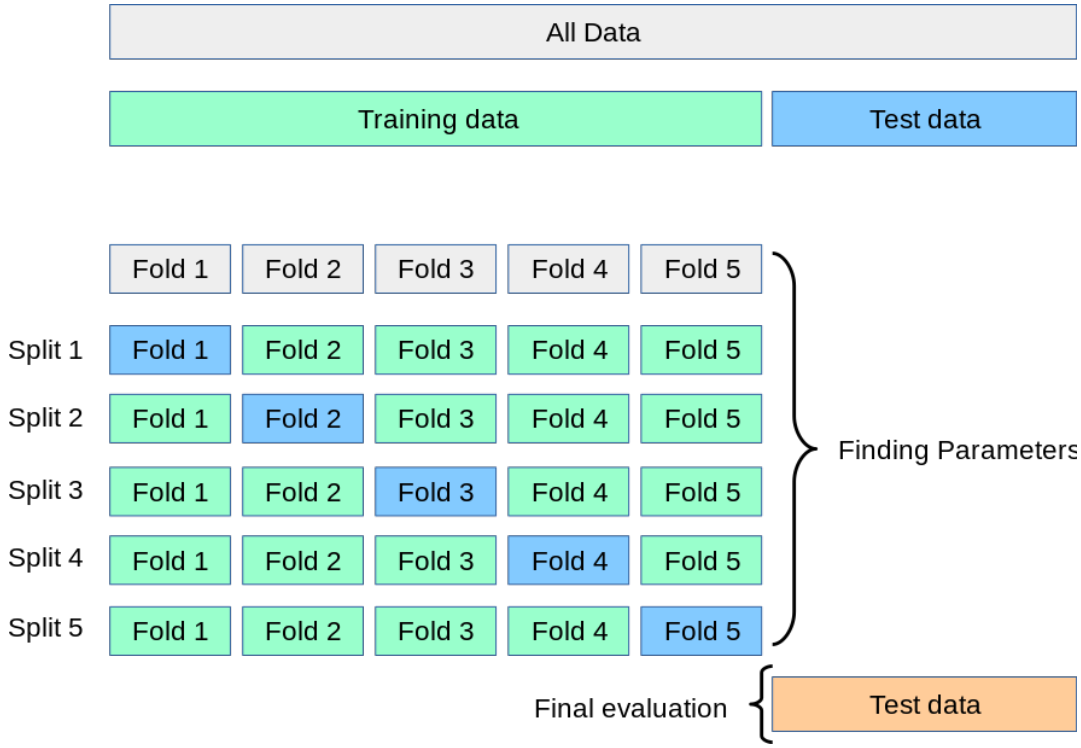
In [ ]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2
0, random_state=1)
```

In [ ]:
```
# Select hyperparam values to try

alphas = [0.01, 0.1, 1] # L1 + L2
l1_ratios = [0.2, 0.5, 0.8] # L1 / L2 ratio

# create all combinations [(0.01, 0.2), (0.01, 0.5), (...)]
import itertools
hyperparams = itertools.product(alphas, l1_ratios)
```

In [ ]:
```
# Train and CV-score model for each combination
from sklearn.linear_model import ElasticNet
from sklearn.metrics import r2_score
from sklearn.model_selection import cross_val_score

for hyperparam in hyperparams:
    alpha = hyperparam[0]
    l1_ratio = hyperparam[1]

    model = ElasticNet(alpha=alpha, l1_ratio=l1_ratio)

    r2 = cross_val_score(model, X_train, y_train, cv=5).mean()

    print(f"alpha: {alpha}, l1_ratio: {l1_ratio},   r2: {r2}")
```

```
alpha: 0.01, l1_ratio: 0.2,   r2: 0.3097093011229858
alpha: 0.01, l1_ratio: 0.5,   r2: 0.36553389455838525
alpha: 0.01, l1_ratio: 0.8,   r2: 0.44169590096847555
alpha: 0.1, l1_ratio: 0.2,   r2: 0.04607452785123458
alpha: 0.1, l1_ratio: 0.5,   r2: 0.08029085880552811
alpha: 0.1, l1_ratio: 0.8,   r2: 0.1778184138979217
alpha: 1, l1_ratio: 0.2,   r2: -0.021420175696800325
alpha: 1, l1_ratio: 0.5,   r2: -0.019482185329917502
alpha: 1, l1_ratio: 0.8,   r2: -0.0114266833108428
```

🔥 Grid Search CV



1. Randomy split your training set into `k` folds of same size
2. Make fold `#1` a val_set, train model on other `k-1` folds & mesure val_score
3. Make fold `#2` a val_set and repeat
4. ...
5. Compute average val_score over all folds

☝️ This is your cross-validated score for **one** given set of hyper-parameters

- Repeat for each value of hyper-param to test
- Save the test set for final evaluation only (AFTER hyper-params are chosen)

Welcome to Sklearn `GridSearchCV` [(https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.htm](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) 🚀

In [ ]:
```
from sklearn.model_selection import GridSearchCV

# Train/Test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
20, random_state=1)

# Instantiate model
model = ElasticNet()

# Hyperparameter Grid
grid = {
    'alpha': [0.01, 0.1, 1],
    'l1_ratio': [0.2, 0.5, 0.8]
}

# Instantiate Grid Search
search = GridSearchCV(
    model,
    grid,
    scoring = 'r2',
    cv = 5,
    n_jobs=-1 # parallelize computation
)

# Fit data to Grid Search
search.fit(X_train, y_train);
```

In [ ]:
```
# Best score
search.best_score_

# Best Params
search.best_params_

# Best estimator
search.best_estimator_
```

Out[ ]:
```
▾ ElasticNet
ElasticNet(alpha=0.01, l1_ratio=0.8)
```

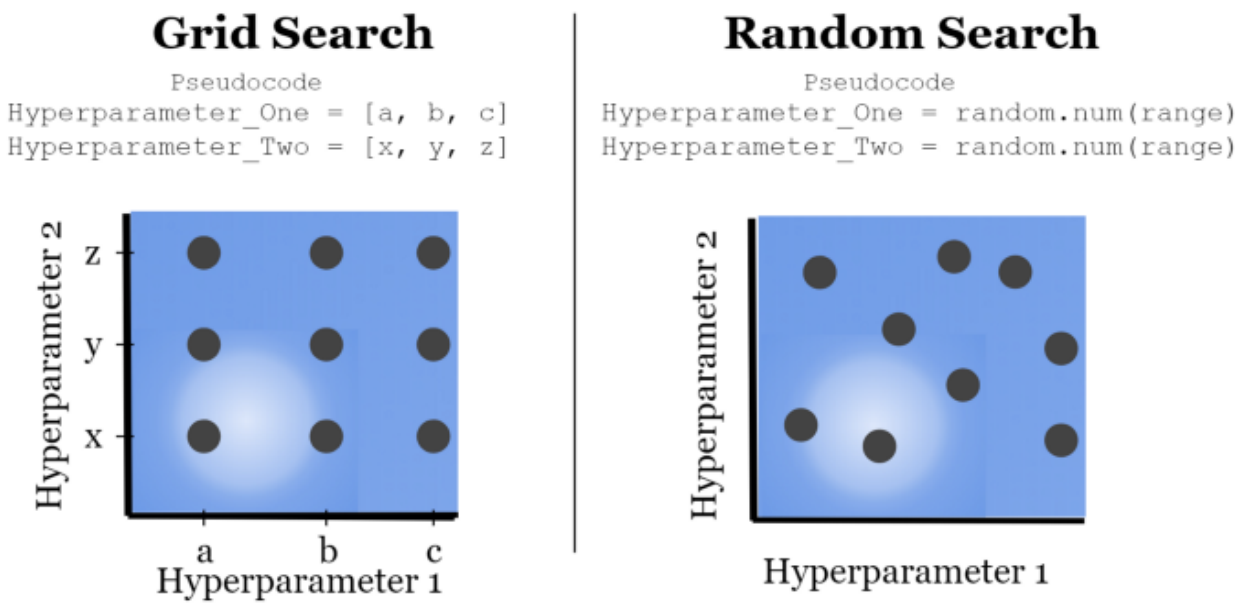
👎 Limitations of Grid Search:

- Computationally costly
- The optimal hyperparameter value can be missed
- Can overfit hyperparameters to the training set if too many combinations are tried out for too small a dataset

Random Search

Randomly explore hyperparameter values from:

- A hyperparameter space to randomly sample from
- The specified number of samples to be tested



Sklearn's `RandomizedSearchCV`

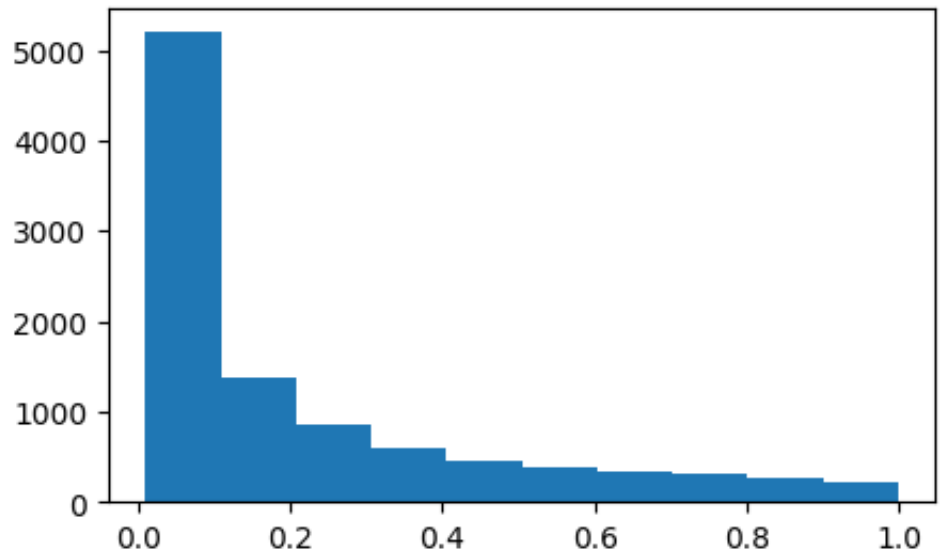
In [ ]:
```
from sklearn.model_selection import RandomizedSearchCV
from scipy import stats

# Instantiate model
model = ElasticNet()

# Hyperparameter Grid
grid = {'l1_ratio': stats.uniform(0, 1), 'alpha': [0.001, 0.01, 0.1,
1]}

# Instantiate Grid Search
search = RandomizedSearchCV(
    model,
    grid,
    scoring='r2',
    n_iter=100,  # number of draws
    cv=5, n_jobs=-1
)

# Fit data to Grid Search
search.fit(X_train, y_train)
search.best_estimator_
```

Out[ ]:
```
▾ ElasticNet
ElasticNet(alpha=0.001, l1_ratio=0.7098199517233674)
```

Choose hyperparameter probability distribution wisely

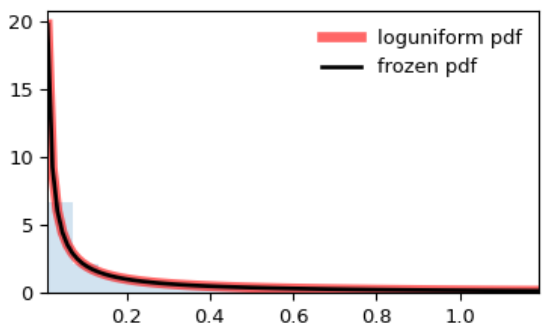
Can be generated with [scipy.stats.distributions (https://docs.scipy.org/doc/scipy/reference/stats.html)](https://docs.scipy.org/doc/scipy/reference/stats.html)

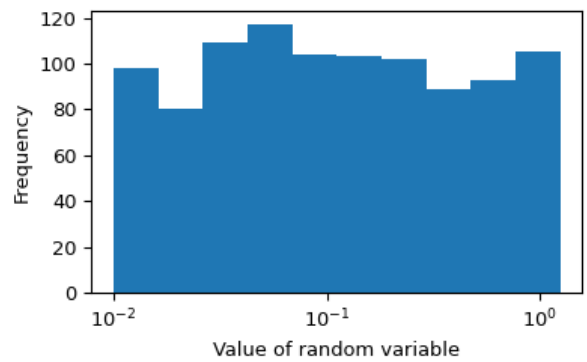
In [ ]:
```
from scipy import stats

plt.figure(figsize=(5, 3))

dist = stats.norm(10, 2) # if you have a best guess (say: 10)

dist = stats.randint(1,100) # if you have no idea
dist = stats.uniform(1, 100) # same

dist = stats.loguniform(0.01, 1) # Coarse grain search

r = dist.rvs(size=10000) # Random draws
plt.hist(r);
```



`loguniform` is great for coarse-grain search across several orders of magnitude

e.g. `loguniform(0.01, 1)` search over [
10^{-2}
,
10^{-1}
,
10^{0}
]



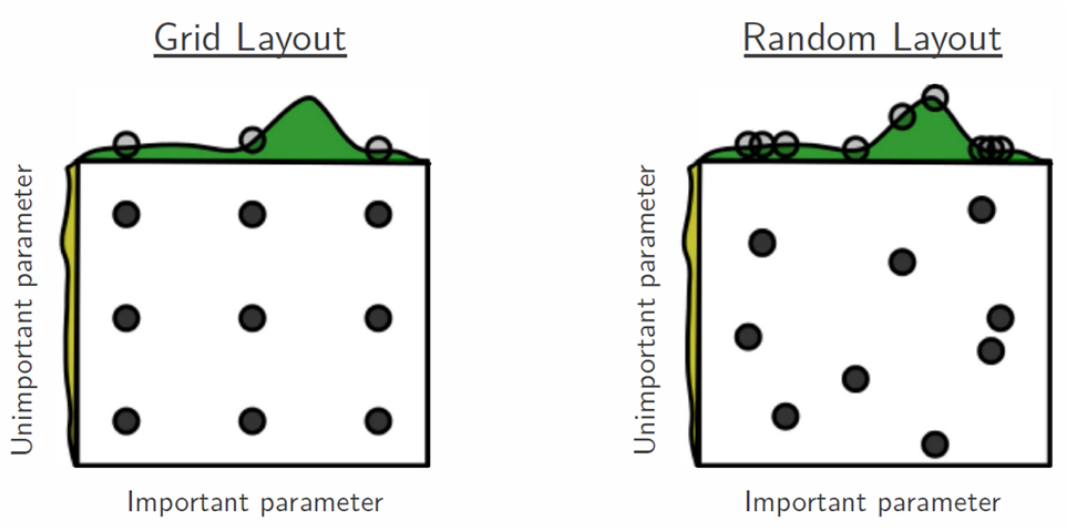


[Doc (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.loguniform.html)](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.loguniform.html)

`RandomizedSearch` vs `GridSearch`

👉 Randomized Search:

- Less typing, if you want to try many values
- Control for the number of combinations to try & time spent searching
- Useful when some hyper-parameters are more important than others



In any case:
✅ Always start with a coarse grain approach (can use Grid or RandomSearch)
✅ Then afterward, fine-tune your search

🔥 Key Summary 🔥

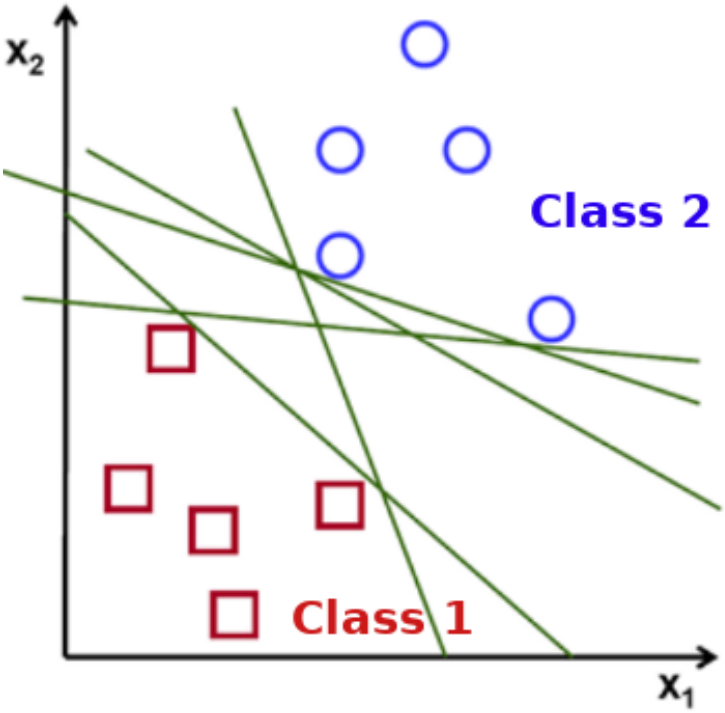
FIT = finding the best **PARAMETERS** that minimize the **LOSS**

FINE-TUNE = finding the best **HYPERPARAMETERS** that maximize **PERFORMANCE METRICS**

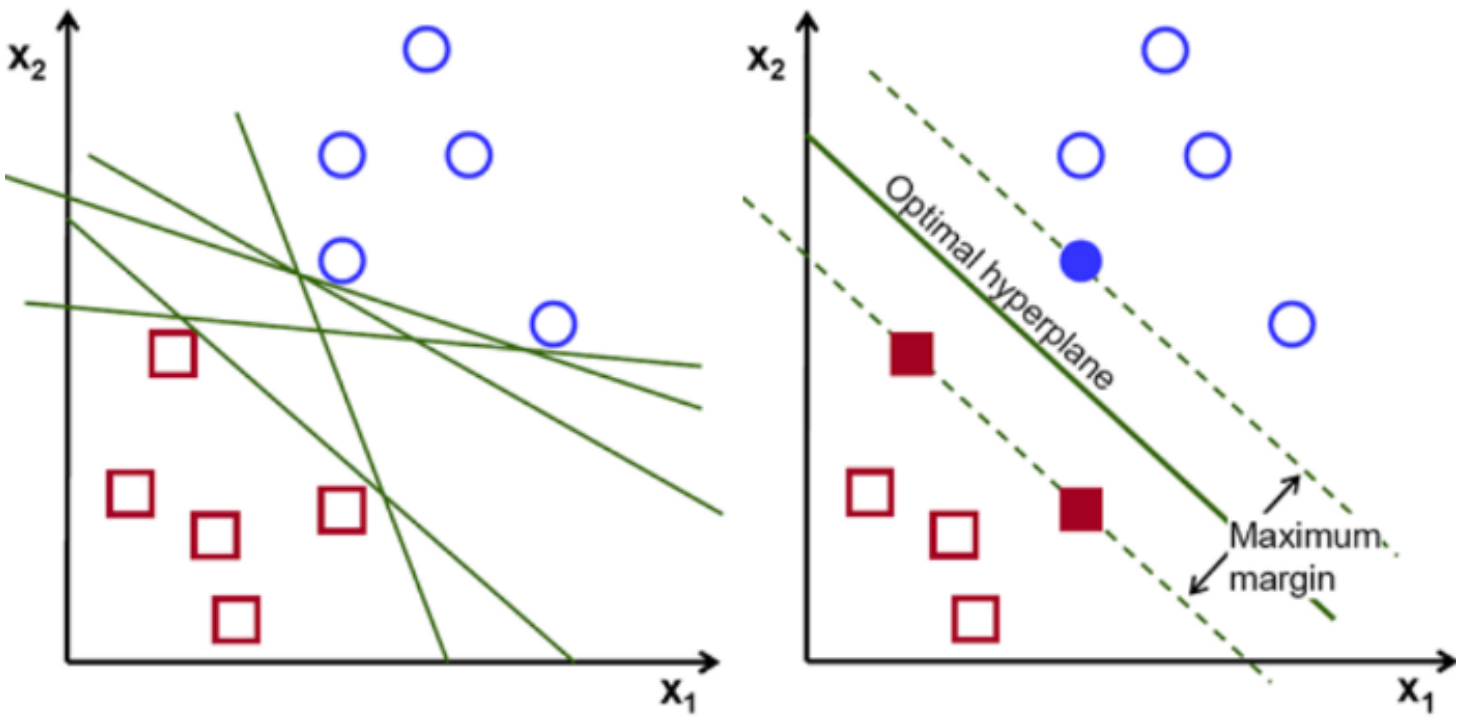
---

4. Support Vector Machines (SVMs)

What's a good decision boundary for classification?



Infinite number of potential decision boundaries that separate the classes ("hyperplanes")

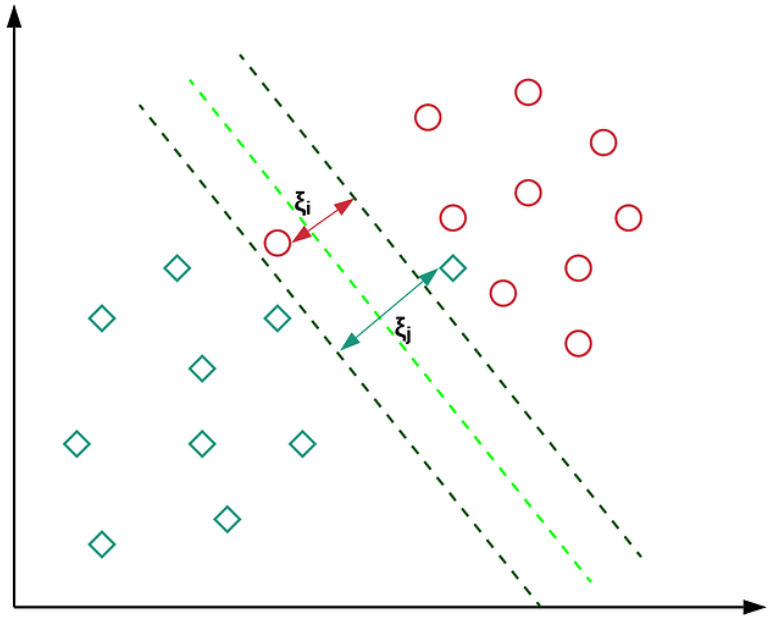


- The hyperplane that generalizes best to unseen data is the one that is furthest from all the points (maximizes the **margin**)
- The points on the margin boundary are called **support vectors**
- Finding them is a convex optimization problem (one single best solution)
- --> **Maximum Margin Classifier** algorithm

🤔 When would such a model be problematic?

- Max Margin is super sensitive to outliers
- It **overfits** to the training data

For **generalization** purposes, we may want to allow some points to be **inside** the margin, or even **on the other side** of the decision boundary:



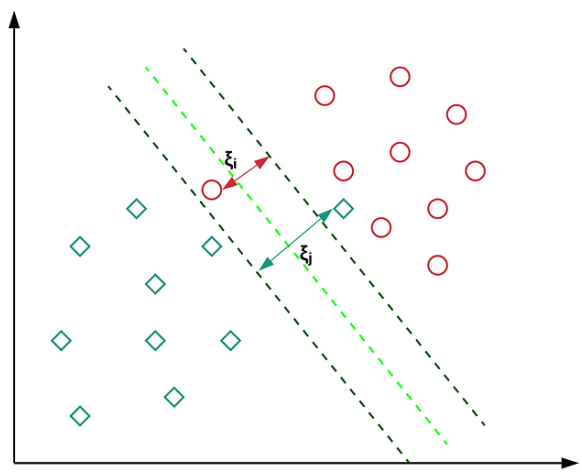
👉 Soft Margin Classifier

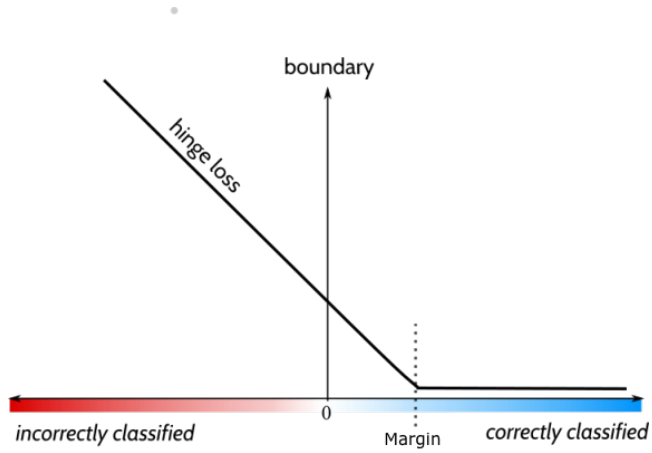
Allows a few points to be misclassified but with a **penalty(**
ξ
)

Penalty for how "far" (
ξ
) they lie on the wrong side of the margin

The **Hinge Loss** is the penalty applied to each point on the wrong side

- The deeper a point lies within the margin, the higher the loss
- The penalty is linear, like MAE





❓ **How strong** should the penalty be for wrongly classified datapoints?
$\leftrightarrow$
How steep should the hinge loss be?
$\leftrightarrow$
How narrow should the margin be?

Tradeoff between classifying training data well and generalizing to new data

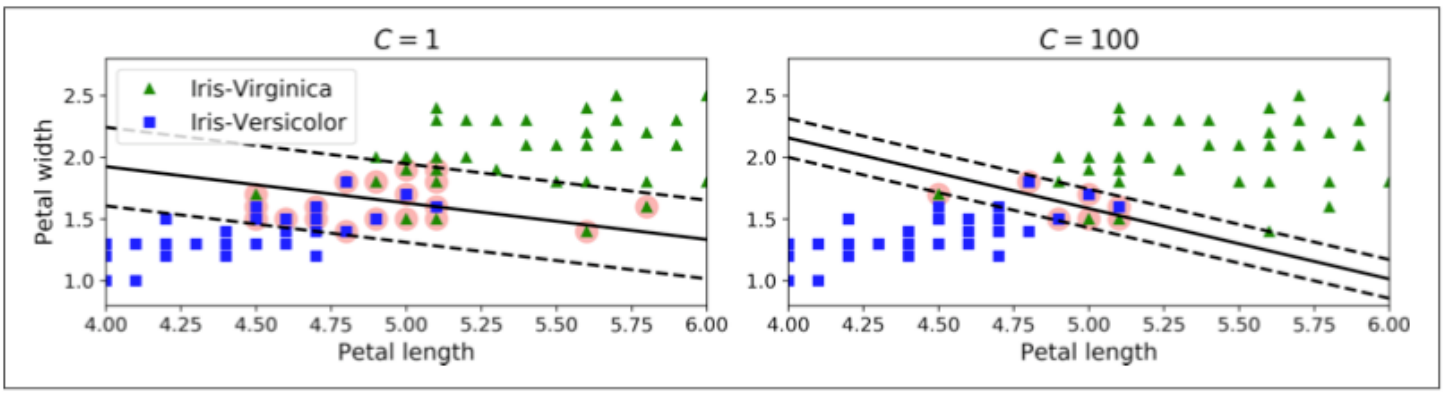
Bias vs. Variance tradeoff again!

- Solution? **Regularization**

Regularization hyperparameter `C`

Strength of the penalty applied on points located on the wrong side of the margin

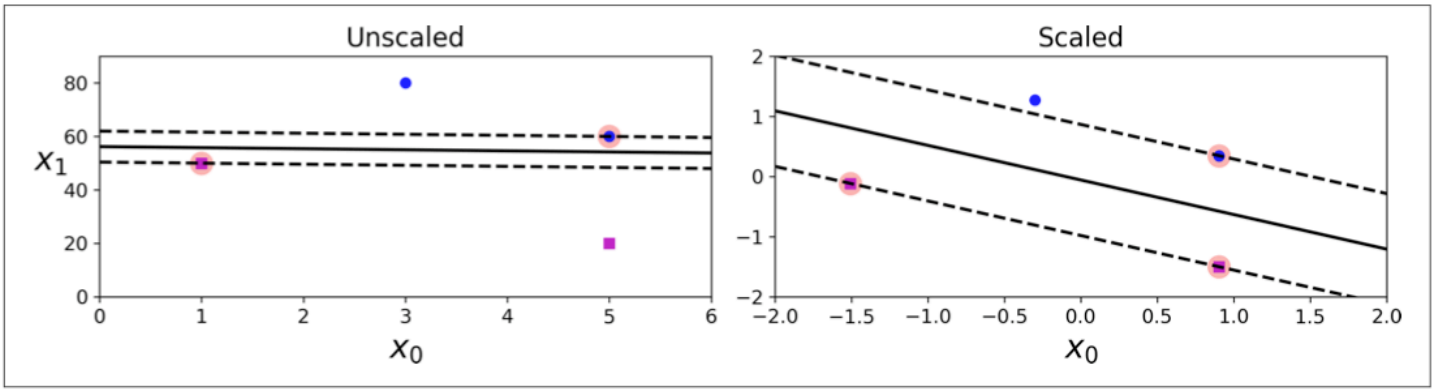
- The higher `C`, the stricter the margin
- A "maximum margin classifier" has `C` = $+\infty$
- The smaller `C`, the softer the margin, the more it is **regularized**
- C is similar to $1/$ in Ridge



💻 sklearn implementation

In [ ]:
```
from sklearn.svm import SVC
svc = SVC(kernel='linear', C=10)

# equivalent but with SGD solver
from sklearn.linear_model import SGDClassifier
svc_bis = SGDClassifier(loss='hinge', penalty='l2', alpha=1/10)
```

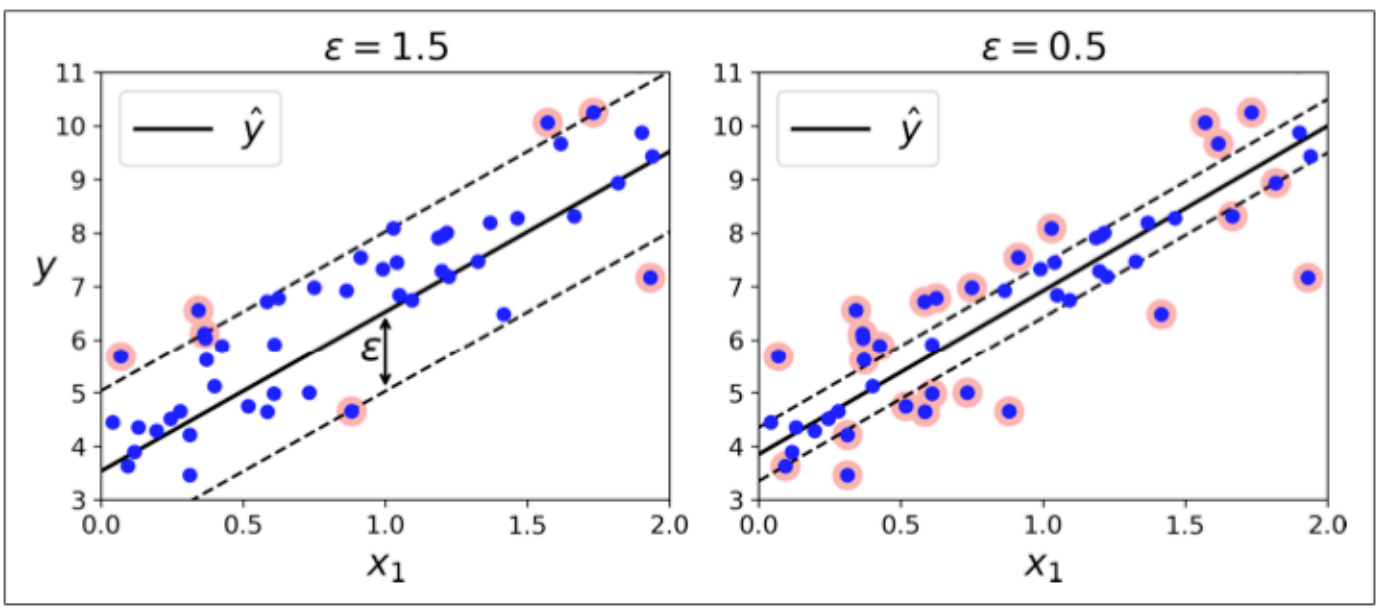
⚠️ Warning: All support vector models require **scaling**



(Bonus) SVM Regressors

The trick is to reverse the objective:

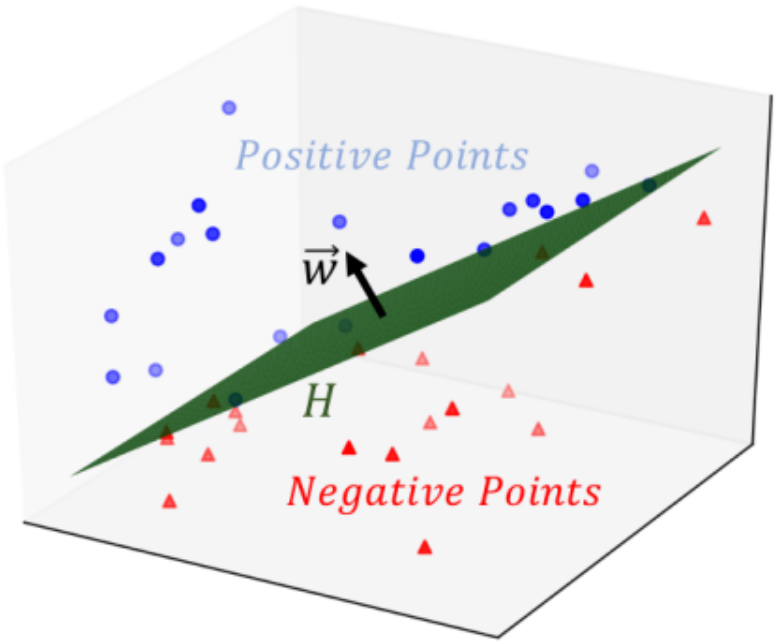
- **Classification**: fit the largest possible *street* **between** two classes
- **Regression**: fit as many points as possible **within** the *street*
- Width of the street controlled by an additional hyperparam `epsilon` ϵ




```
from sklearn.svm import SVR
regressor = SVR(epsilon=0.1, C=1, kernel='linear')
```

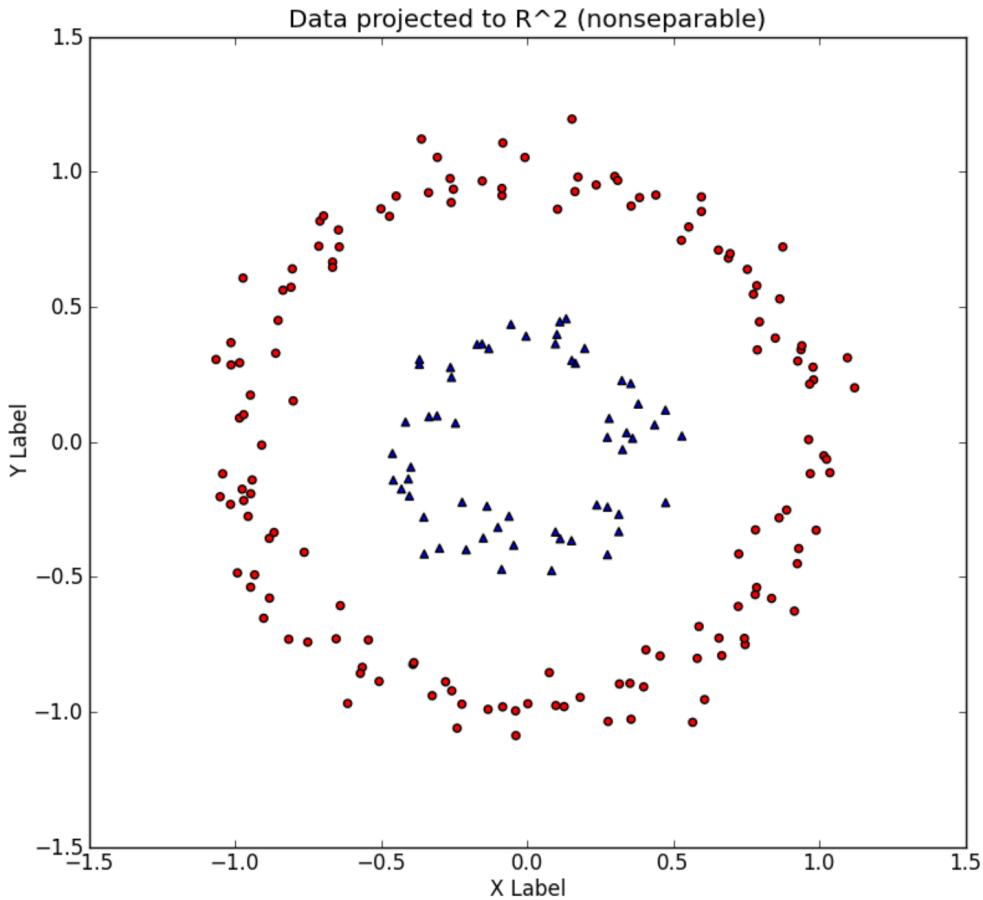
5. SVM Kernels (Keep for Recap 🕛)

Fitting a **Linear SVM** is finding the best vector **w**



- whose **direction** uniquely determines the decision boundary hyperplane (orthogonal)
- which minimizes the sum of **hinge losses** for outliers

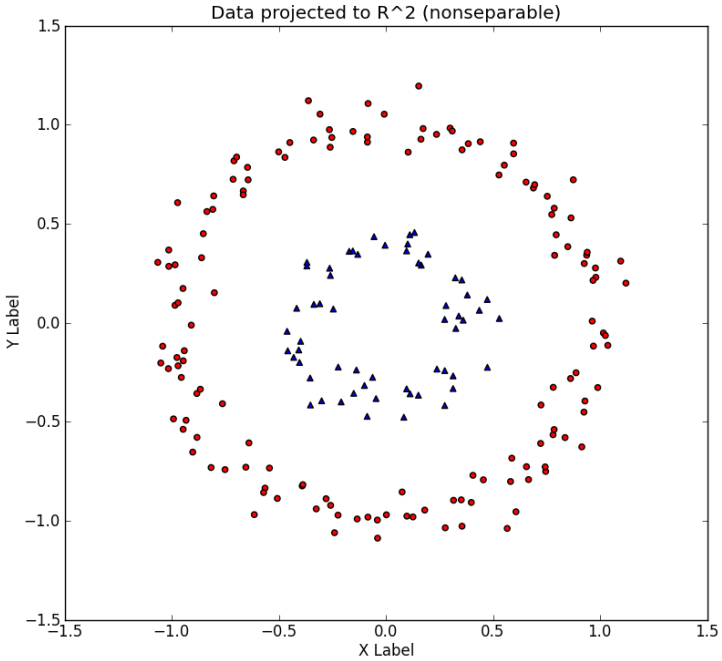
What about this?

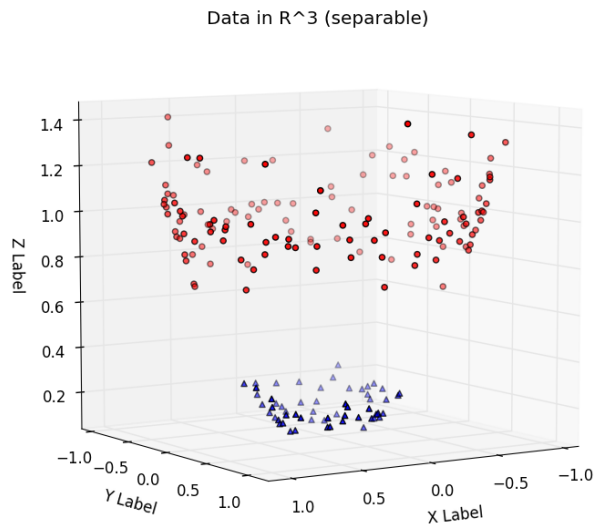


Not linearly separable!

We could add a new feature
$Z = (X^2 + Y^2)$

In a higher dimension, the data becomes **linearly** separable again!





What we just did is a **feature mapping**
ϕ
from 2D to 3D

$$\phi(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 + x_2^2 \end{bmatrix}$$

More precisely, a polynomial mapping of degree `d=2`

🤯 **Problem:** we have increased the dimensionality of our feature space!

Running an expensive SVM in higher dimensions can become extremely intensive

The Kernel Trick 🔥

Instead of explicitly creating all the new features, smart people came up with a very clever "trick":

- Each time the loss function is calculated, it calculates a sort of **similarity** $K(\mathbf{a}$ between all pairs of data points, called a **Kernel**
- Two points with large a similarity would be classified similarly
- We can **simulate** feature mapping by wisely replacing the Kernel of the Loss Function
- Much more computationally **efficient**

📚 [Read more (https://xavierbourretsicotte.github.io/Kernel_feature_map.html)](https://xavierbourretsicotte.github.io/Kernel_feature_map.html)

List of SVM Kernels

`kernel` specifies the type of **feature mapping** to be used to make data **linearly separable** again

- `linear`
- `poly` (of dimension `d`)
- `rbf` (radial basis function (https://en.wikipedia.org/wiki/Radial_basis_function) of coef `gamma`)
- `sigmoid` (of coef `gamma`)

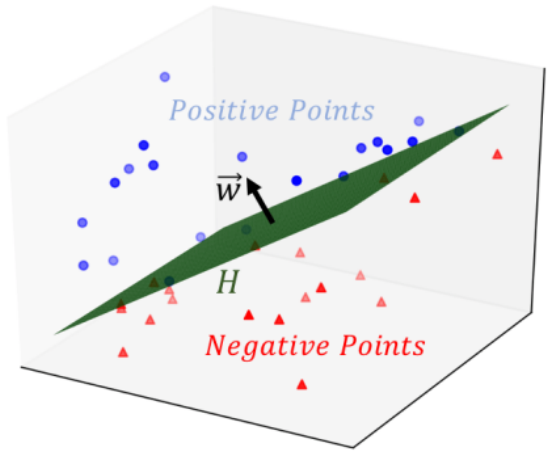
`C` is the **strength** of the cost associated with the **wrong classification**

5.2 Kernel details

a) Linear Kernel

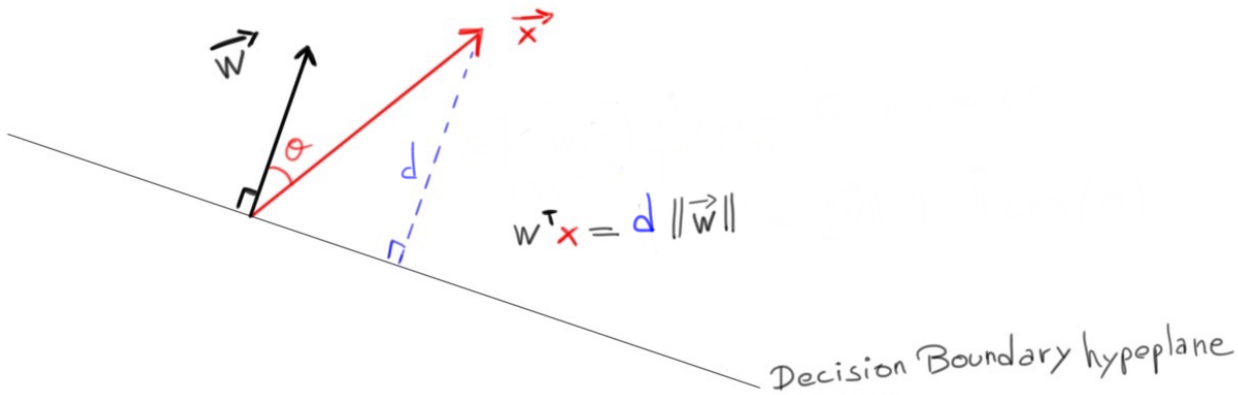
This is the hypothesis function
$h_{\mathbf{w}}(X)$
of a Linear SVM

$$h_{\mathbf{w}}(X) = \begin{cases} 0 \text{ if } \mathbf{w}^T\mathbf{x} & < -1 \\ 1 \text{ if } \mathbf{w}^T\mathbf{x} & \geq 1 \end{cases}$$



☝️ We say that the **Kernel** of Linear the SVM is
$K(\mathbf{a}$

$K(\mathbf{a}$



💡 Notice that
$\mathbf{w}^T\mathbf{x}$
is proportional to the perpendicular distance
d
to the decision boundary hyperplane

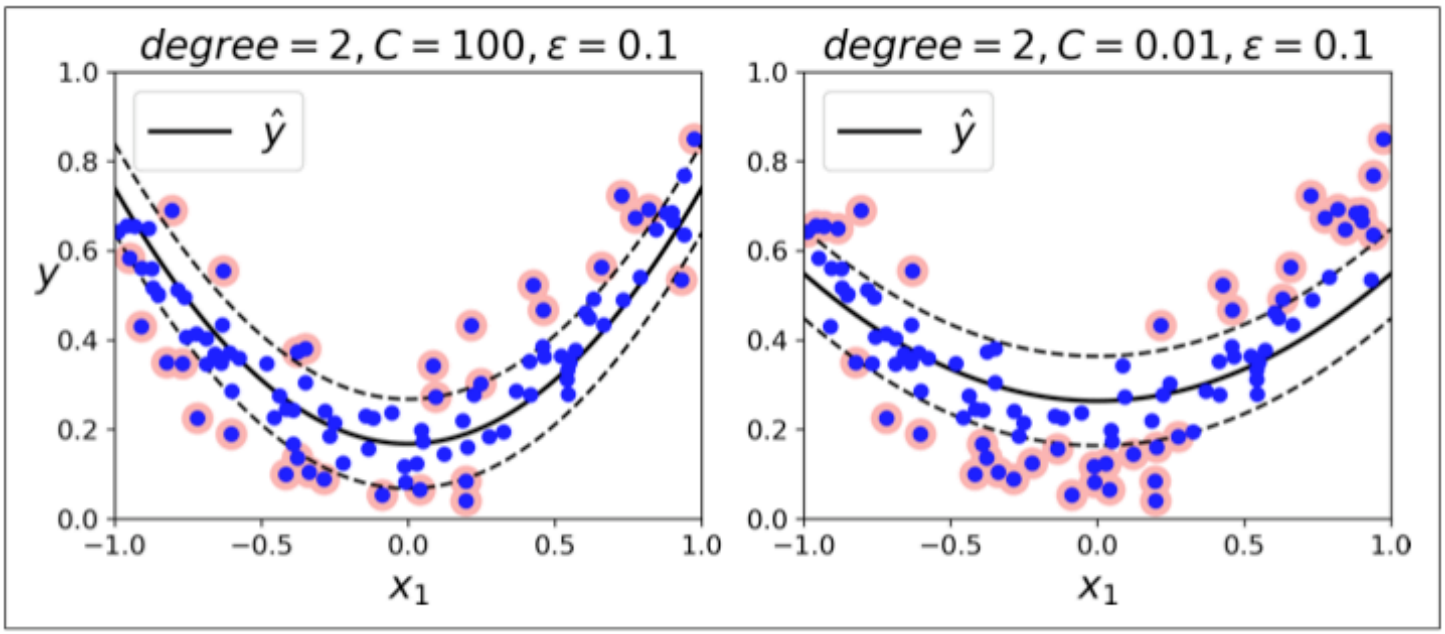
b) Polynomial Kernel (order 2)

$K(\mathbf{a}$

Polynomial Kernel order of d

$K(\mathbf{a}$

The polynomial kernel also allows fitting non-linear **regressions** very easily




```
regressor = SVR(epsilon=0.1, C=1, kernel='poly', degree=2)
```

Source: [Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/)](https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/)

c) The RBF Kernel (aka Gaussian)

👇 *exponentially decreasing value of the distance between the two points a and b*

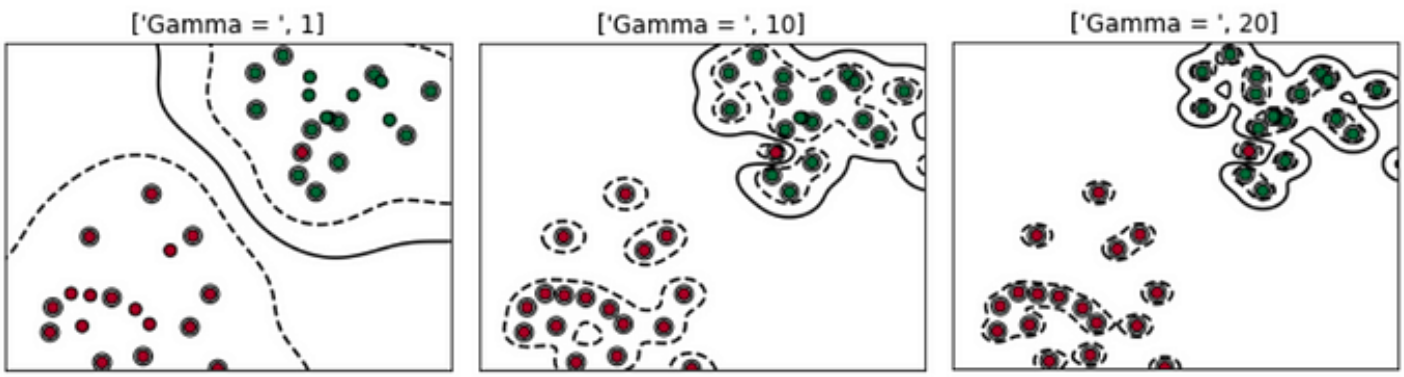
$K(\mathbf{a}$

Similarity between two datapoints is "gaussian"

Two points far away from one another are exponentially more likely to be different.

γ
acts as a *myopia* factor.

👉 **Increasing**
γ
makes model overfit



Recommended reads

- 📚 Hands-on Machine Learning with Sklearn (2020), Chapter 5 SVM Section "under the hood"
- Kernels explained (https://xavierbourretsicotte.github.io/Kernel_feature_map.html) (Math)
- SVM vs Logistic Regression (http://www.cs.toronto.edu/~kswersky/wp-content/uploads/svm_vs_lr.pdf) (Math)

Your turn! 🚀