CITS5508 Machine Learning

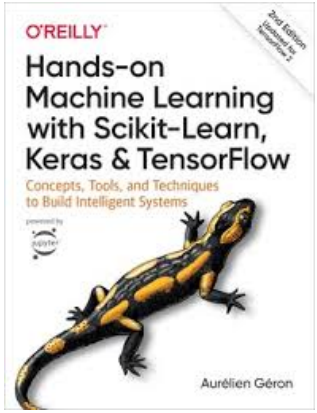
Débora Corrêa (Unit Coordinator and Lecturer)

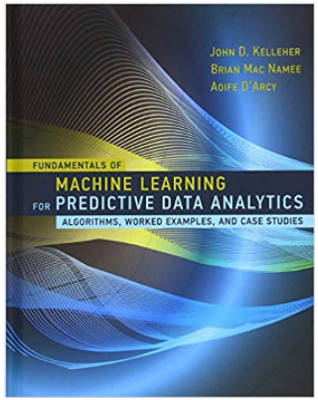
2023

Today

More about chapters 3 and 4.
k-Nearest Neighbours (k-NN)

Hands-on Machine Learning with Scikit-Learn & TensorFlow
Fundamentals of Machine Learning for Predictive Data Analytics (chapter 5)





Summary

We will continue to look at how Machine Learning algorithms work, discuss about some important aspects and introduce the k-Nearest Neighbours (k-NN) algorithm. We will cover:

- The Bias x Variance tradeoff
- Cross-validation and Grid-search
- Early Stopping
- Regularised Linear Models
- Softmax Regression
- k-Nearest Neighbours (k-NN) algorithm

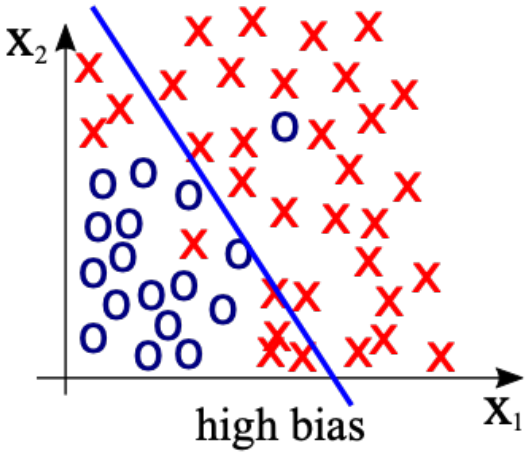
Underfitting the Training Data

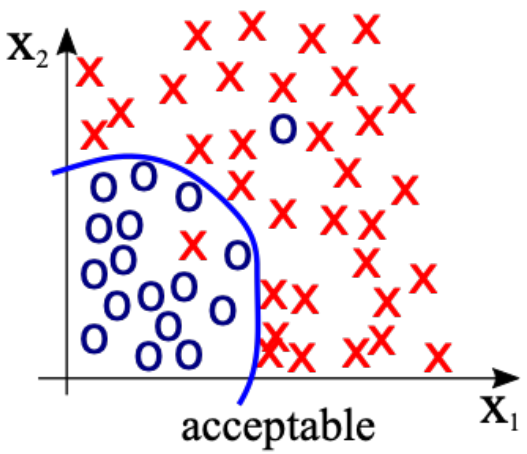
Underfitting occurs when your model is too simple to learn the underlying structure of the data.

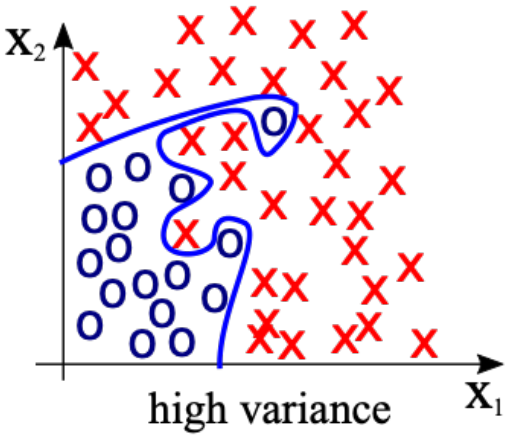
The main options to fix this problem are:

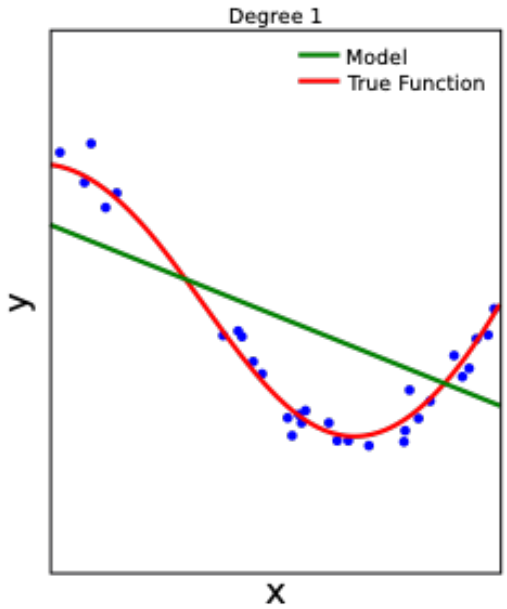
- Selecting a more powerful model, with more parameters
- Feeding better features to the learning algorithm (feature engineering)
- Reducing the constraints on the model (e.g., reducing the regularization hyperparameter)

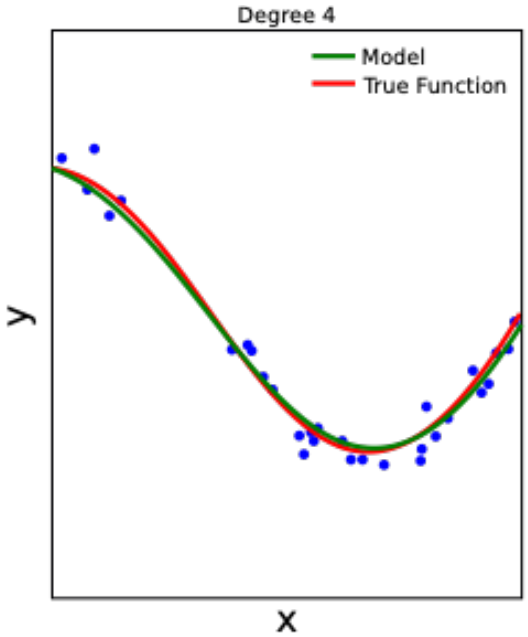
Underfitting and Overfitting

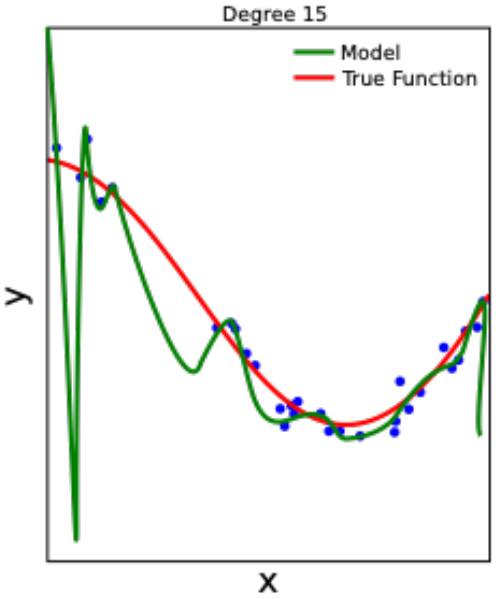












Bias/Variance Tradeoff

High bias: performance on the training set

A larger or new set of features may help.

ML models with higher variance (more complex) may help.

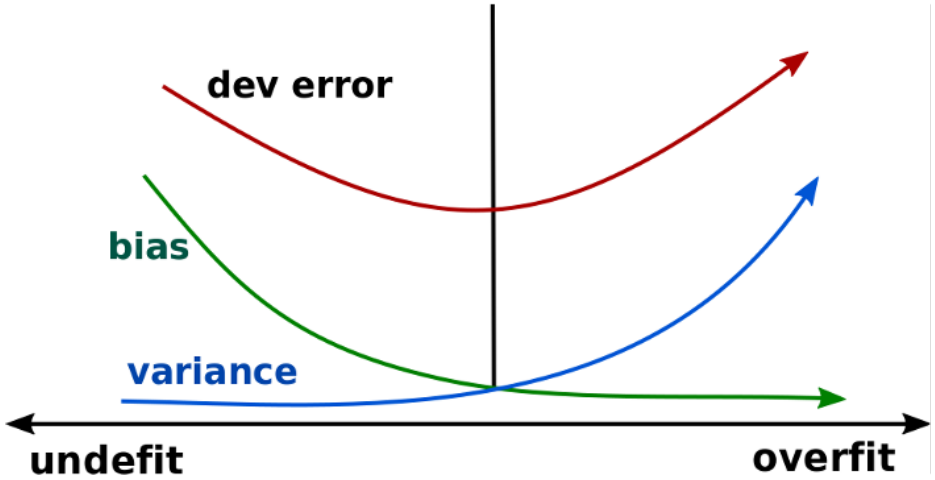
High variance: performance on the validation set

Getting more training instances may help.

Smaller set of features may help.

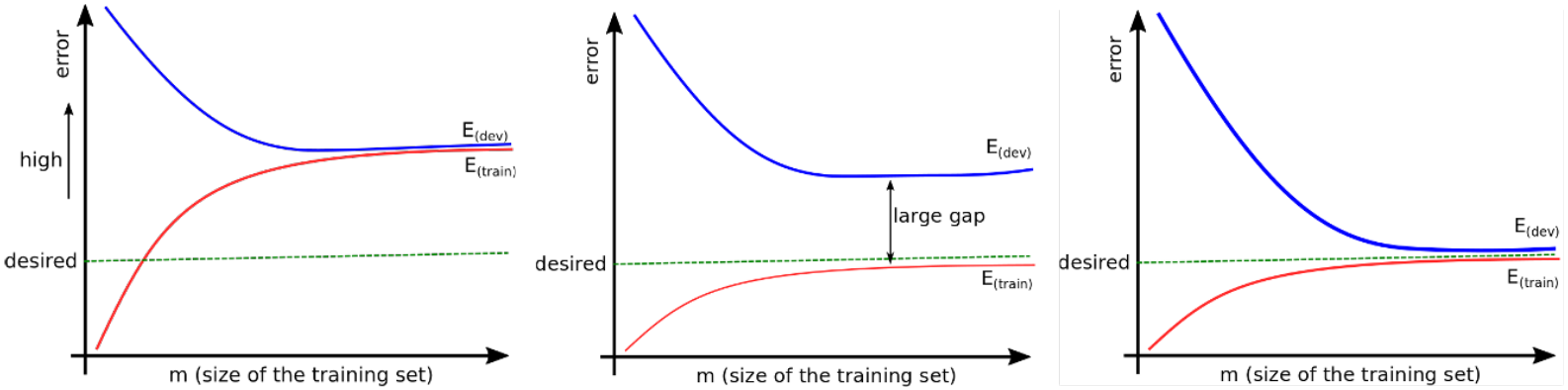
Decrease the complexity of the model.

Regularisation techniques.

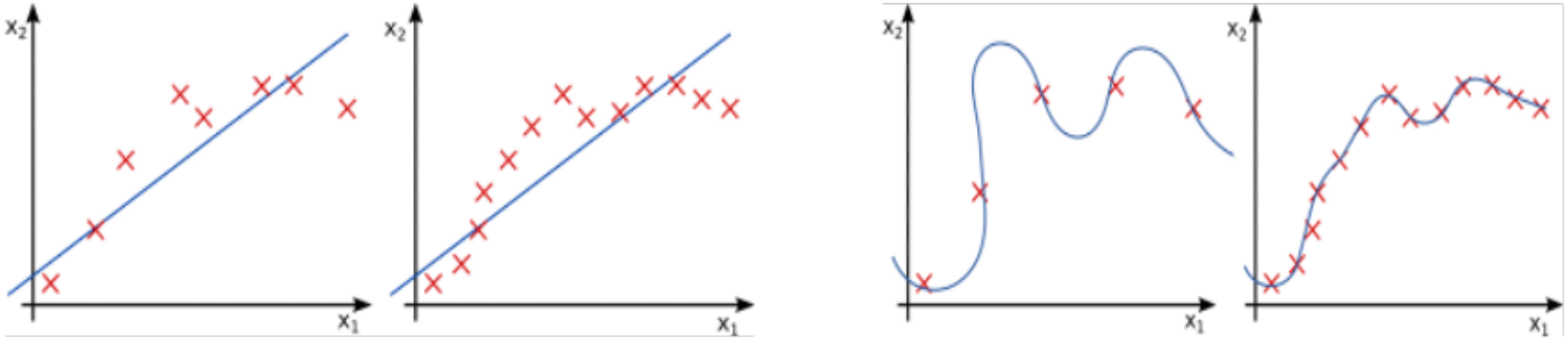


Learning Curves

Inspect the model performance (validation and training sets) as the number of training examples increases



Underfitting: more training instances may not help.
Overfitting: more training instances may help.



Bias/Variance Tradeoff

A model's generalization error can be expressed as the sum of 3 different errors:

Bias – this part of the generalization error is due to wrong assumption, e.g. assumption that the data is linear when it is quadratic. A high bias model tends to give underfitting problem.

Variance – this part of the generalization error is due to the model's excessive sensitivity to small variation in the training data. A high variance model tends to give overfitting problem.

Irreducible error – this part is due to the natural variability of the data.

- Increasing a model's complexity will typically increase its variance and reduce its bias.
- Reducing a model's complexity increases its bias and reduces its variance.

Fine-Tuning

Tuning hyperparameters is an important part of building a Machine Learning system.

A hyperparameter is a parameter of a learning algorithm (not of the model). It is not affected by the learning algorithm itself, and it must be set prior to training and remains constant during training.

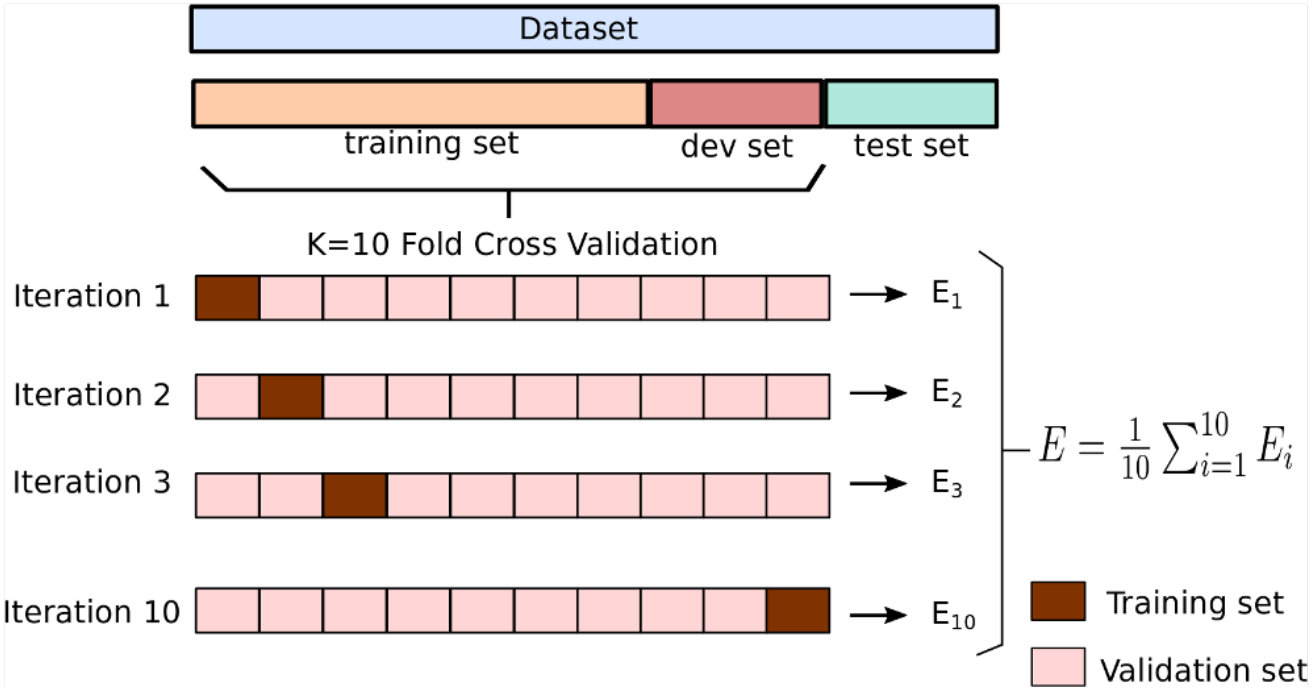
Examples: the value k of the k-NN algorithm, the degree of a Polynomial Regression, the regularisation hyperparameter, the threshold of a Logistic Regression.

Using the test set to pick the best hyperparameter values tends to make the model not perform well new data other than the test set, we should also need a validation set.

K Fold Cross-validation

Train multiple models with various hyperparameters on the reduced training set (i.e., the full training set minus the validation set), and you select the model that performs best on the validation set.

The training set is split into K subsets (folds). For each K iteration, the model is trained and validated using a different combination of such sets.



K Fold Cross-validation

The error rate on the test set is called the generalization error (or out-of-sample error).

Grid Search (find good combination of hyperparameter values). Use Scikit-Learn's `GridSearchCV` to do this via cross-validation.

Or use `RandomizedSearchCV` instead when the hyperparameter search space is large.

After this holdout validation process, you train the best model on the full training set (including the validation set), and this gives you the final model. Lastly, you evaluate this final model on the test set to get an estimate of the generalization error.

Early Stopping

Another way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called *early stopping*.

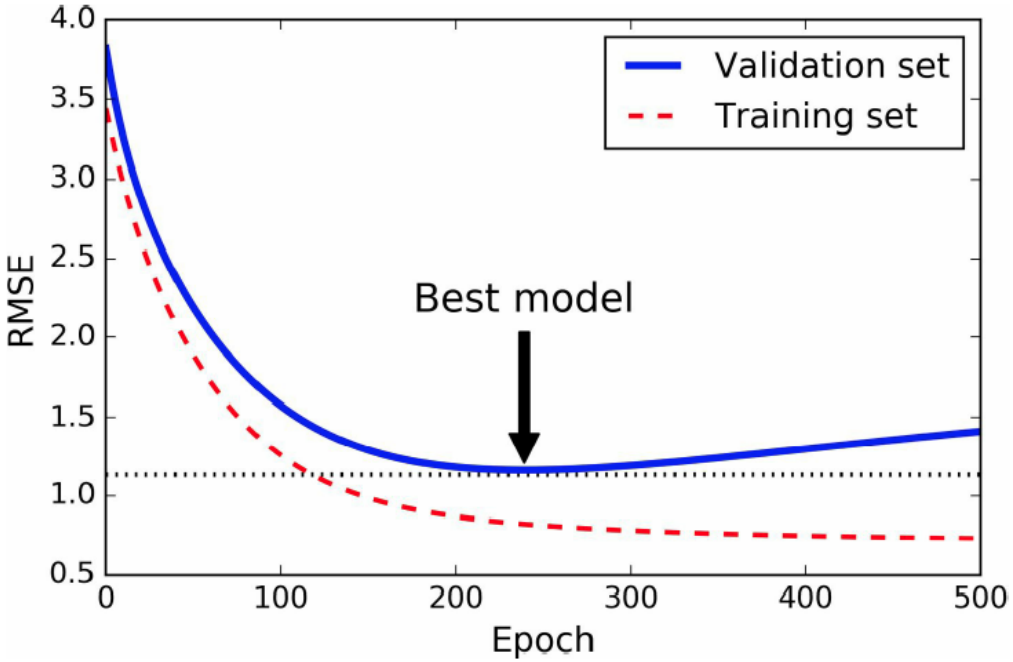


Figure 4-20. Early stopping regularization

(Note: with stochastic and mini-batch gradient descent, the curves are not so smooth.)

Early Stopping

```python
from copy import deepcopy

# prepare the data
poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
    ])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = deepcopy(sgd_reg)
```

Regularised Linear Models

A good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. For example, a simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model. We will firstly look at three different ways to constrain the weights:

- Ridge Regression
- Lasso Regression
- Elastic Net

Ridge Regression

Ridge Regression (also called *Tikhonov regularization*) is a regularized version of Linear Regression, with the regularization term equalling to

$$\alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

which is added to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. Note that the regularization term should only be added to the cost function during training.

Once the model is trained, you evaluate the model's performance using the unregularized performance measure.

The Ridge Regression cost function is:

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Ridge Regression

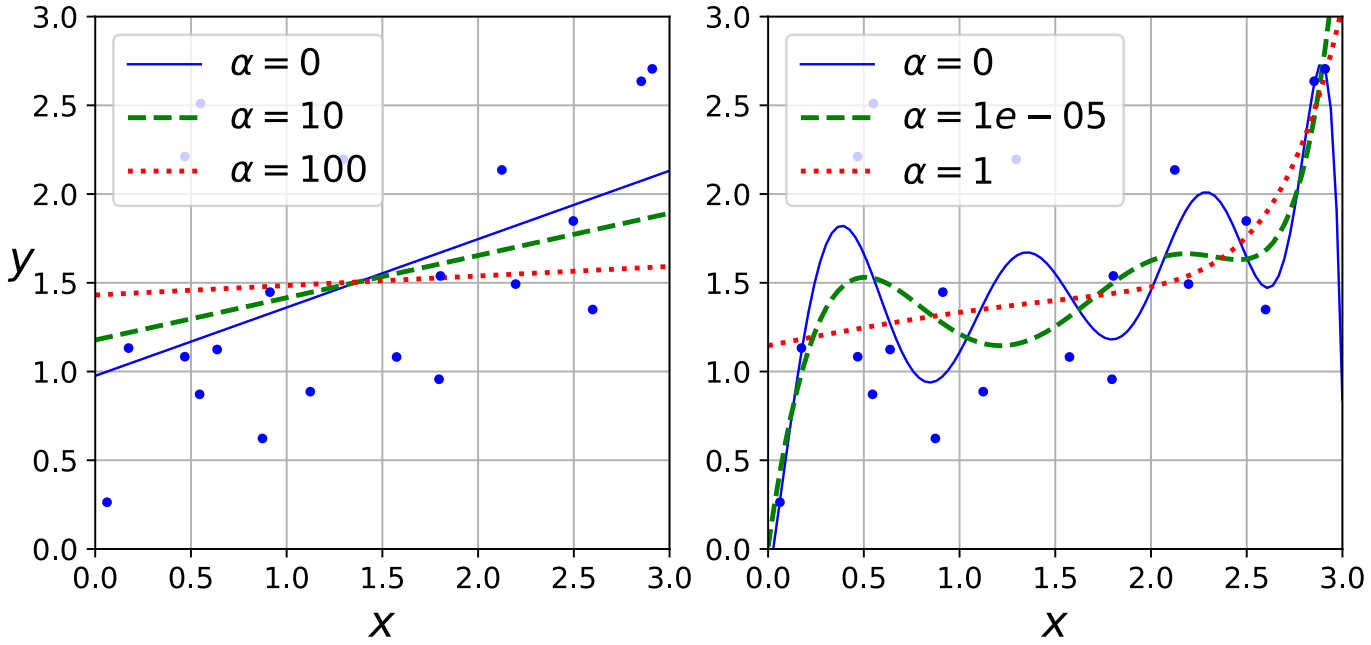


Figure 4-17. Ridge Regression

Left: Using ridge regression to fit a linear model (1^{st} degree polynomial); Right: using ridge regression to fit a 10^{th} degree polynomial.

See the `sklearn.linear_model.Ridge` class. Can also use `sklearn.linear_model.SGDRegressor` with `penalty='ℓ2'`.

Lasso Regression

Uses $|\theta_i|$ instead of θ_i^2 for the regularization term. So the Lasso Regression cost function is: $J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{n} |\theta_i|$

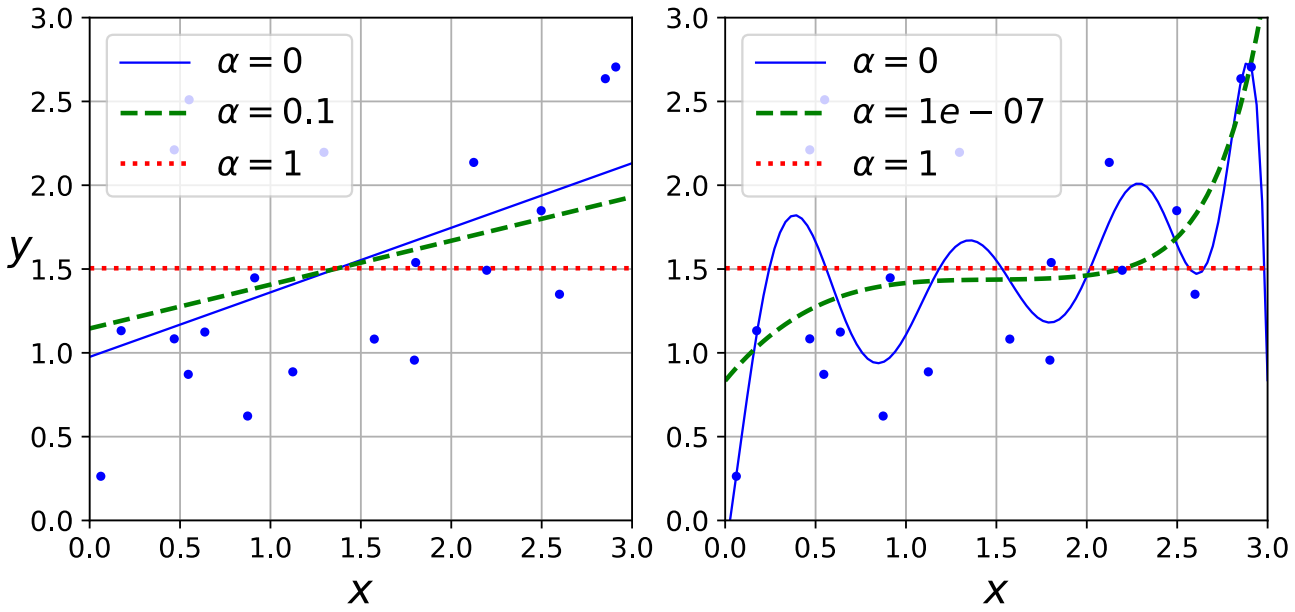
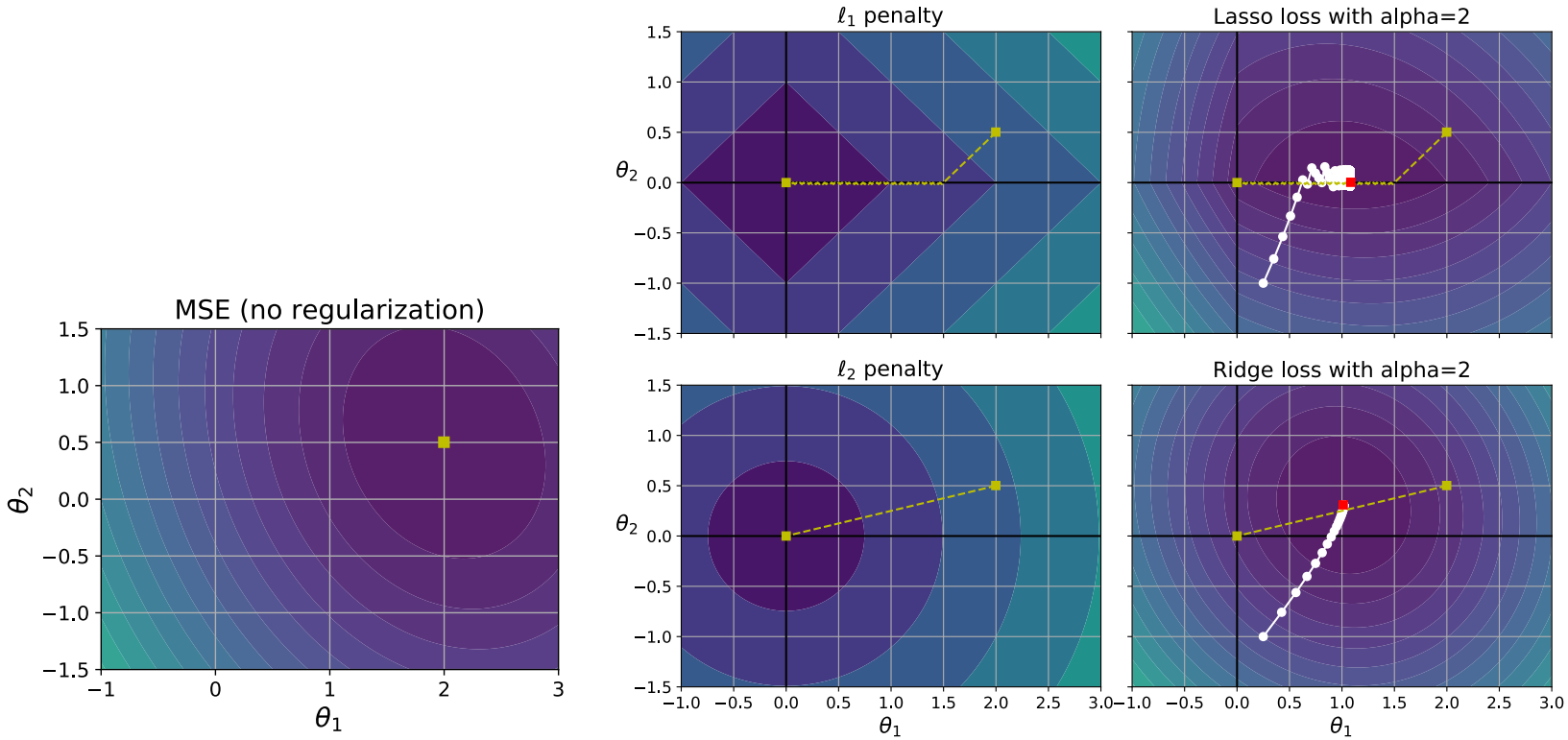


Figure 4-18. Lasso Regression

Left: Using lasso regression to fit a 1st degree polynomial;
Right: using lasso regression to fit a 10th degree polynomial.
See the `sklearn.linear_model.Lasso` class. Can also use `sklearn.linear_model.SGDRegressor` with `penalty='ℓ1'`.

Lasso regularization vs Ridge regularization



Optimal parameters for the unregularized MSE are: $\theta_1 = 2$ and $\theta_2 = 0.5$

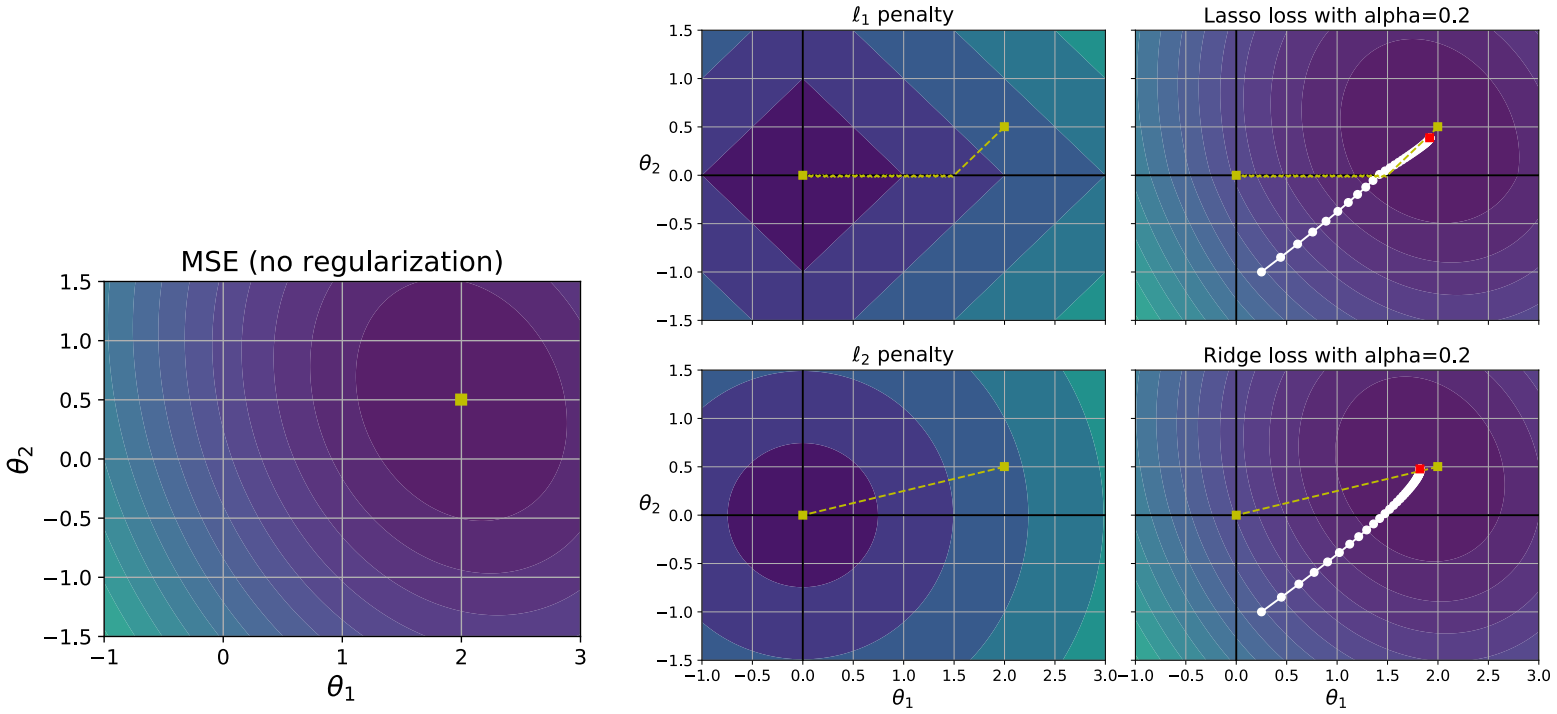
Middle column: contour plots of ℓ_1 and ℓ_2 penalties: $|\theta_1| + |\theta_2|$ and $(\theta_1^2 + \theta_2^2)/2$

Right column: contour plots of Ridge loss (MSE plus ℓ_1 penalty) and Lasso loss (MSE plus ℓ_2 penalty). Regularization coefficient α is set to 2.

– The optimal parameters obtained from Lasso loss are: $\theta_1 = 1.084$ and $\theta_2 = 0.003$.

– The optimal parameters obtained from Ridge loss are: $\theta_1 = 1.0120$ and $\theta_2 = 0.310$.

Lasso regularization vs Ridge regularization



Optimal parameters for the unregularized MSE are: $\theta_1 = 2$ and $\theta_2 = 0.5$

Middle column: contour plots of ℓ_1 and ℓ_2 penalties: $|\theta_1| + |\theta_2|$ and $(\theta_1^2 + \theta_2^2)/2$

Right column: contour plots of **Ridge loss** (MSE plus ℓ_1 penalty) and **Lasso loss** (MSE plus ℓ_2 penalty). Regularization coefficient α is set to 0.2.

– The optimal parameters obtained from Lasso loss are: $\theta_1 = 1.918$ and $\theta_2 = 0.388$.

– The optimal parameters obtained from Ridge loss are: $\theta_1 = 1.822$ and $\theta_2 = 0.478$.

Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. It combines both regularization terms together. The Elastic Net cost function is:

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

where we now have hyperparameters α and r as two regularization coefficients.

See the `sklearn.linear_model.ElasticNet` class.

Softmax Regression

Logistic Regression can be generalized to support multiple classes directly. This is called *Softmax Regression* or *Multinomial Logistic Regression*.

Given an instance $\mathbf{x}$, the Softmax Regression model first computes a score $s_k(\mathbf{x})$ for each class k, then estimates the probability $\hat{p}_k$ of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores:

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^{K} \exp(s_j(\mathbf{x}))}$$

where K is the total number of classes and $s_k(\mathbf{x}) = \left(\boldsymbol{\theta}^{(k)}\right) \mathbf{x}$.

Softmax Regression

Softmax Regression will predict the class with the highest estimated probability, that is, the class with the highest score.

$$\hat{y} = \arg\max_k \sigma(\mathbf{s}(\mathbf{x}))_k = \arg\max_k s_k(\mathbf{x}) = \arg\max_k \left(\left(\boldsymbol{\theta}^{(k)} \right) \mathbf{x} \right)$$

Softmax Regression - Cost Function

Similarly to the Logistic Regression, the objective of training the model is to estimate a high probability for the target class (and therefore a low probability for the other classes).

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log\left(\hat{p}_k^{(i)}\right)$$

$y_k^{(i)}$ is the target probability that the i^{th} instance belongs to class k. In general, it is either equal to 1 (instance belongs to the class) or 0 (otherwise).

Decision boundary for the Logistic Regression...

Last week we used the Logistic Regression classifier on two features: *petal length* and *petal width*.

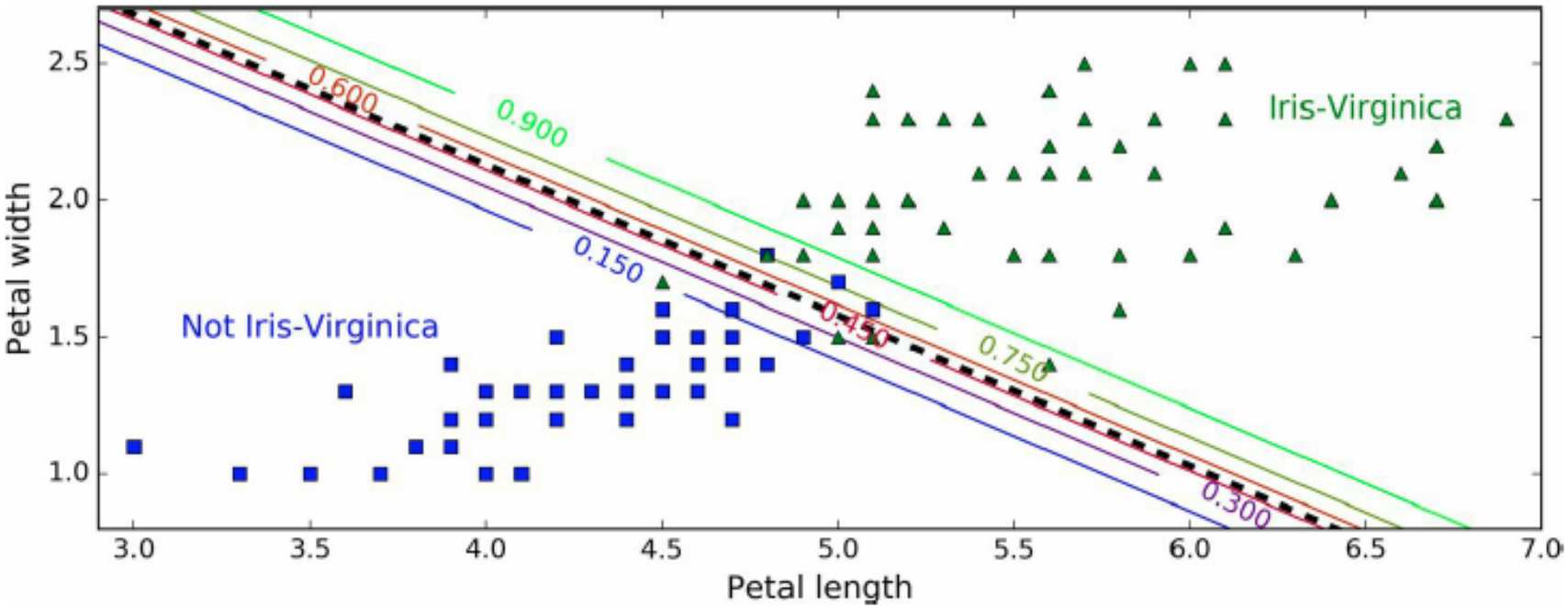


Figure 4-24. Linear decision boundary

The black dashed line represents the model's decision boundary (50% probability)

Applying softmax to the Irises

For the same two features (petal length and petal width), the diagram shows the resulting decision boundaries, represented by the background colours.

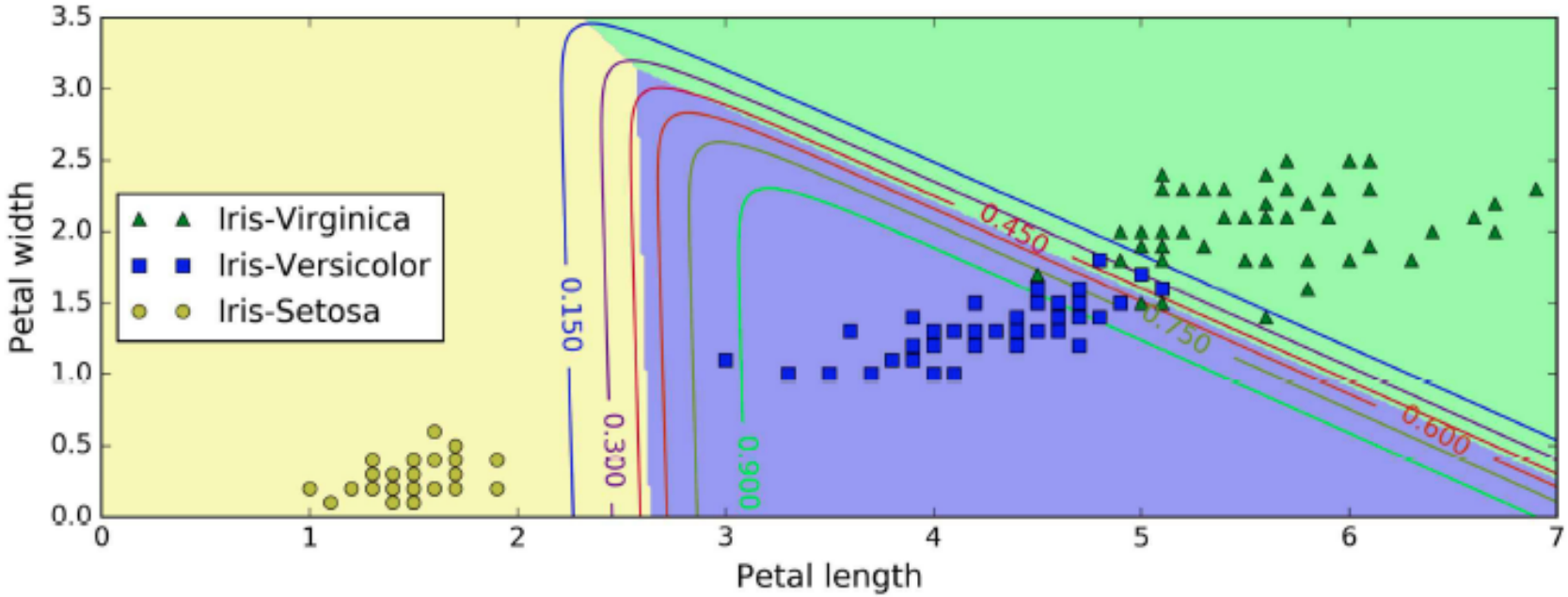


Figure 4-25. Softmax Regression decision boundaries

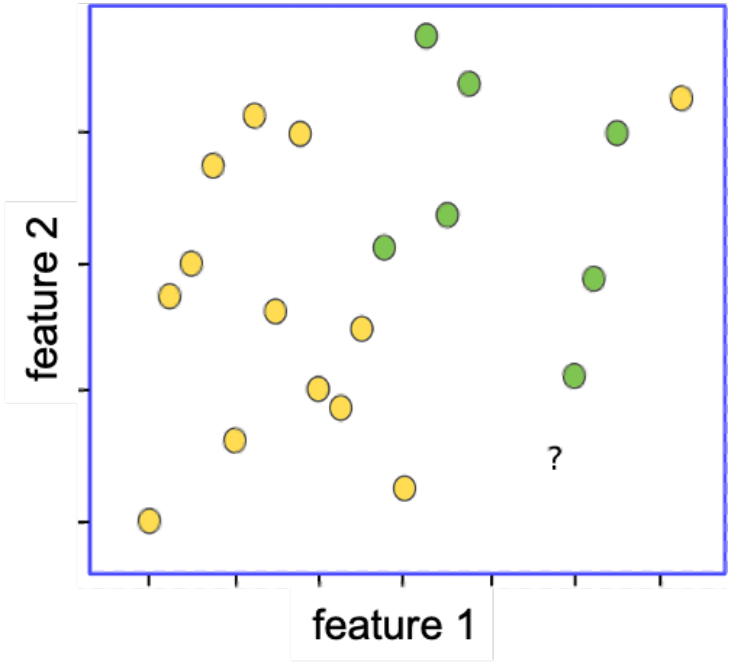
k-Nearest Neighbours (k-NN)

Instance-based learning - a model is not explicitly learned.

Basic idea: similar instances will be closer to each other in the feature space.

Based on measures of similarity: the distance between the instances in the feature space is defined by a distance metric.

Used for classification and regression.



k-Nearest Neighbours (k-NN) - Cont.

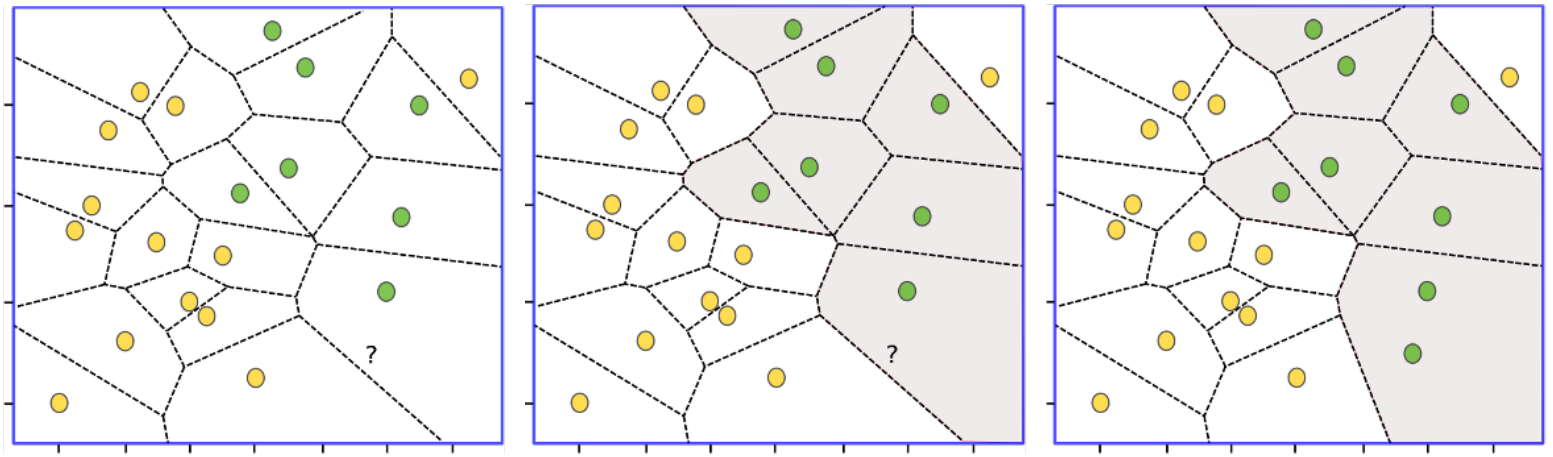
Minkowski distance can be used to calculate the distance between two data instances $\mathbf{x}_i$ and $\mathbf{x}_j$ with n features:

$$D(\mathbf{x}_i, \mathbf{x}_j) = \left(\sum_{i=1}^{n} abs\left(\mathbf{x}_i[i] - \mathbf{x}_j[i]\right)^p \right)^{1/p}$$

The parameter p defines the behaviour of the metric. If $p = 1$ we have the Manhattan distance, if $p = 2$ we have the Euclidean distance.

k-Nearest Neighbours (k-NN) - Cont.

Finding the nearest neighbour (k = 1)



Overall idea: feature space is partitioned locally (similar to a Voronoi tesselation).

Implicitly, a global prediction boundary is determined by aggregating the local partitions within the feature space.

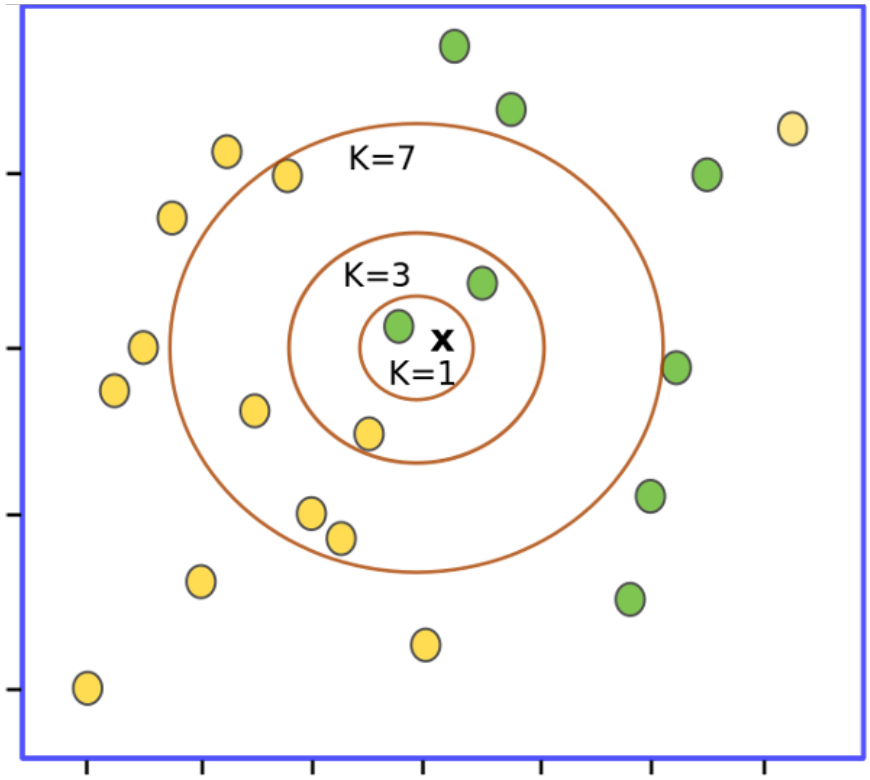
k-Nearest Neighbours (k-NN) - Cont.

We can decrease on individual instances ($k > 1$).

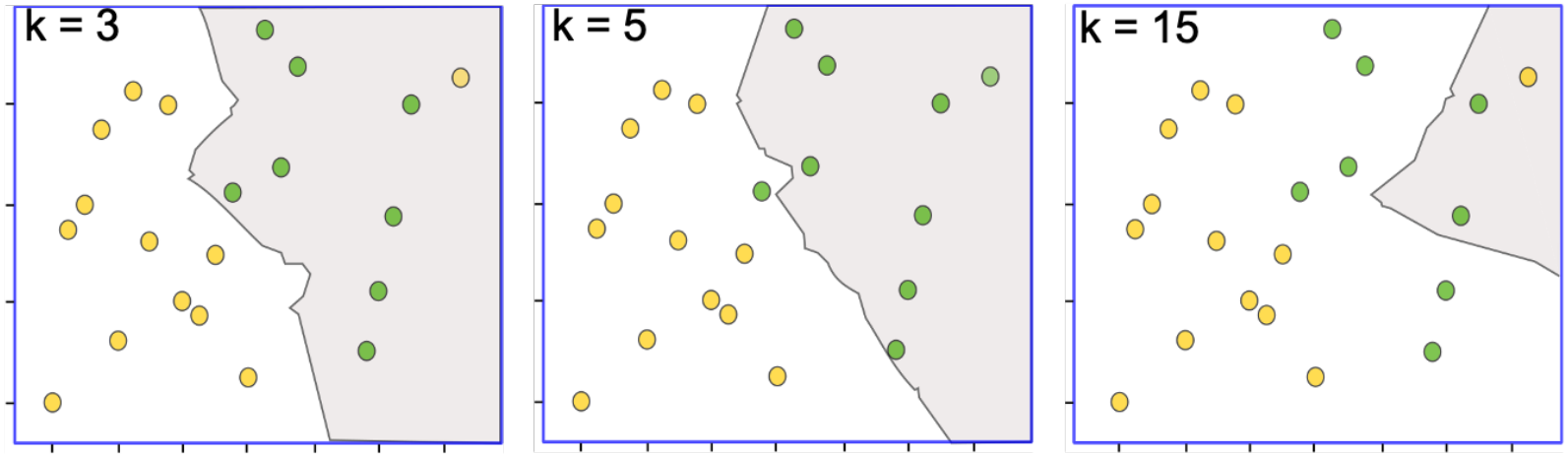
Majority vote in the set of k nearest neighbours.

$$\hat{y}_q = \arg\max_{c \in C} \sum_{i=1}^{k} \delta(c_i, c)$$

where
$\delta(a, b) = 1$ if $a == b$ and
$\delta(a, b) = 0$ if $a \neq b$.



k-Nearest Neighbours (k-NN) - Cont.



For high values of k, there is a tendency towards the majority class, therefore do not work well for imbalanced datasets.

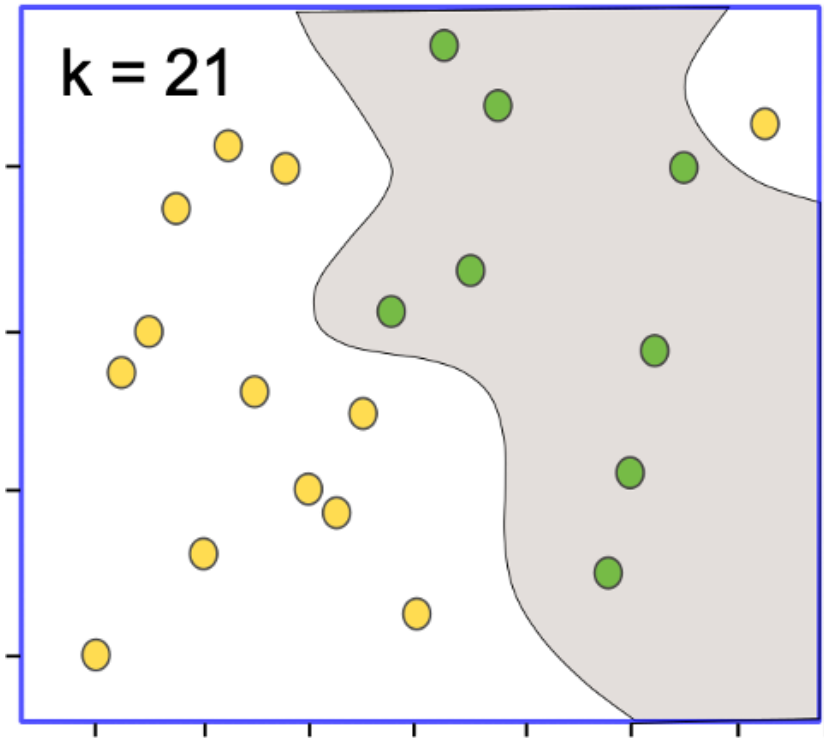
k-Nearest Neighbours (k-NN) - Cont.

Distance weighted k-NN:

Weight the contribution of each neighbour w.r.t. the distance between the query and the neighbour:

$$\hat{y_q} = \arg\max_{c \in C} \sum_{i=1}^{k} w_i \delta(c_i, c)$$

where $w_i = \frac{1}{d(\mathbf{x}_q, \mathbf{x}_i)^2}$



k-Nearest Neighbours (k-NN) - Cont.

Memory-intensive, but simple and intuitive.

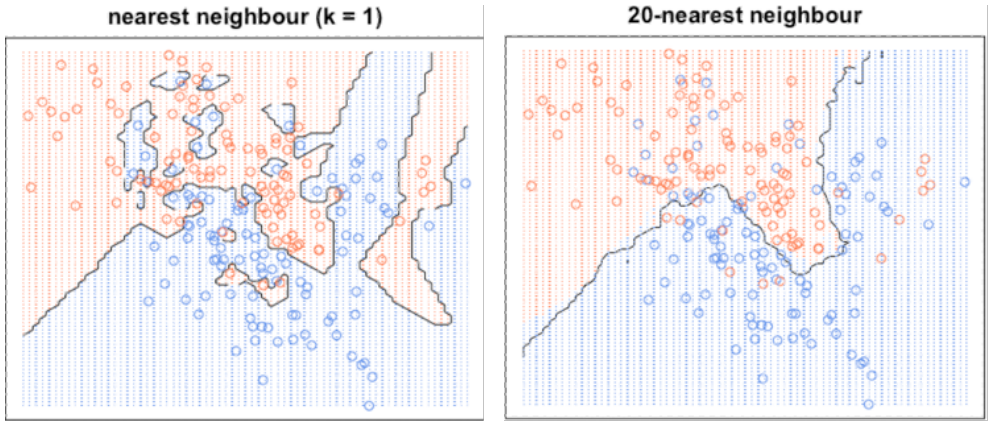
Expensive testing or prediction.

Works better when the density of the feature space is high and similar for each class.

Requires a meaningful distance metric.

Noise and outliers may have negative effect.

Tradeoff: Small values of k: risk of overfitting.
Higher values of k: risk of underfitting.



k-Nearest Neighbours (k-NN) - Cont.

Feature normalisation: the larger the scale the larger the influence of the feature.

Attribute-weighted k-NN:

$$dist_w(\mathbf{x}_i, \mathbf{x}_j) = \sum_{i=1}^{n} w_i \left(x_i[n] - x_j[n]\right)^2$$

May require an expert to advice you or ML to estimate the weights.

Variations:
- Approximate Nearest Neighbour (k-d trees).
- Locally Sensitive Hashing (LSH) - for higher dimensions.
- Density based k-NN.

Multiclass Classification

(As opposed to Binary Classifiers), these are for discriminating between multiple classes ($N > 2$).

Some algorithms (such as the Softmax Regression, Random Forest classifiers or naive Bayes classifiers) are capable of handling multiple classes directly. (Can even get probability of each class).

Others (such as Support Vector Machine classifiers) are strictly binary classifiers.

But you could train N binary classifiers (each class versus the rest) and select the highest score (= *one versus all* or *OvA*).

Alternatively train $(N(N-1)/2)$ to classify between each pair of classes, and see which class wins the most duels (= *one versus one* or *OvO*).

Multilabel Classification

This is the situation where the classifier needs to output multiple class labels for each instance, with each class label taking on binary values (e.g., "yes" / "no", True/False).

Example 1: classifying whether several specific faces are present in each group photo.
Example 2: classifying each document into 4 binary labels: *education*, *health*, *politics*, *religion*.

Can evaluate the performance using F_1 score for each label, then take average.

Multilabel Classification

Example: train a classifier to predict two labels for each input image:
Label 1: whether the digit in the image is larger than 7;
Label 2: whether the digit is odd.

```
from sklearn.neighbors import KNeighborsClassifier
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)

>>> knn_clf.predict([some_digit])  # an image containing digit 5
array([[False, True]])
```

Multioutput-Multiclass Classification

It is simply a generalization of *multilabel classification* where each label can be multiclass (i.e., it can have more than two possible values).

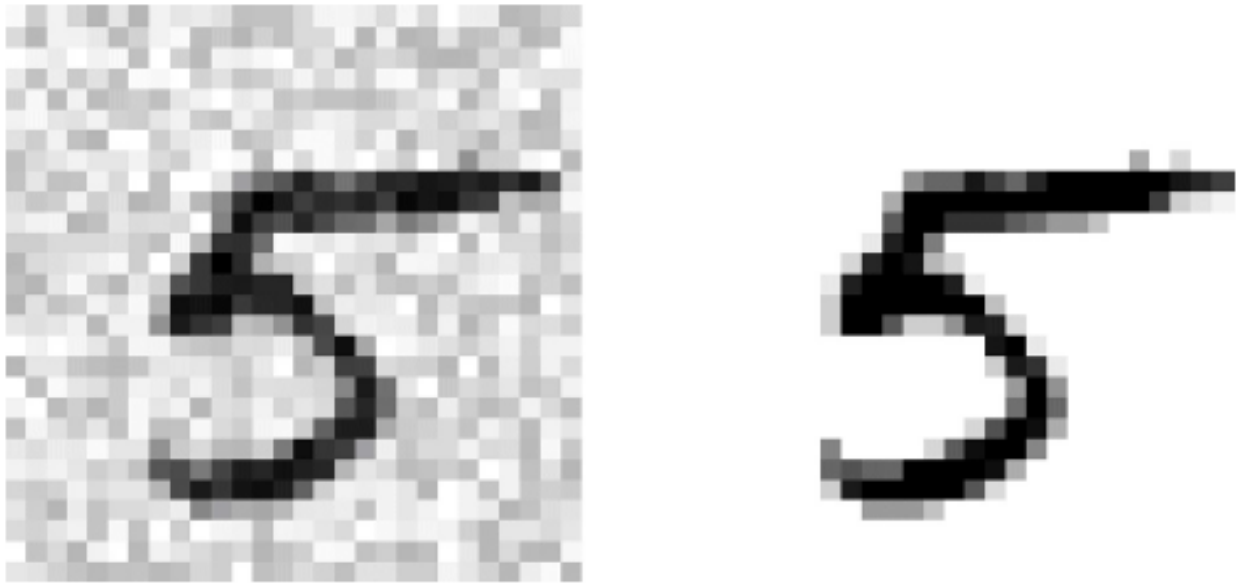
Example 1: classify each image of fruit into two labels:
type: apple, banana, grape, orange, pear (5 classes)
colour: orange, red, green, yellow, purple, brown (6 classes)

Example 2: a system that removes noise from images. Takes as input a noisy 100×100 digit image and outputs a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Each pixel corresponds to one label. The classifier needs to predict $10,000$ labels and each label can have multiple values (pixel intensity ranges from 0 to 255, so 256 classes in total).

Multioutput-Multiclass Classification: Example

Create the training and test sets from MNIST images and adding noise to their pixel intensities using NumPy's `randint()` function. The target images will be the original images.

An example training instance: On the left is the noisy input image, and on the right is the clean target image.



Multioutput-Multiclass Classification

After training the classifier and giving a noisy image as input, this is what we get as output (left).

Another example clean image produced by the classifier is shown on the right.

Multilabel vs Multioutput-Multiclass Classification

	Number of class labels	**Cardinality of each label**
Multiclass classification	1	> 2
Multilabel classification	> 1	2
Multioutput-multiclass classification	> 1	> 2

For next week

Work through your first assignment and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Supplementary material: Derivation of the Normal Equation.

Assignment 1 (10%) requires a submission to *LMS* - check the unit calendar.

Read up to Chapter 5 Support Vector Machines.

And that's all for the fourth lecture.

Have a good week.