Machine Learning

- Infinity and beyond....

Machine Learning

FUNDAMENTALS

- A computer program is said to learn from experience E with respect to some task T, and some performance measure P, if its on T, as measured by P, improved with experience E.

- ML can simplify code and perform.
- Solves complex problem.
- getting insights.

Types of machine learning.

* Supervised, unsupervised, semi supervised and reinforcement learning
* learning incrementally of the fly (online vs batch learning)
* instance based or model based.

Supervised / Unsupervised learning.

- Supervised.

eg Classification

eg To predict a numerical value given a set of feature (mileage, age, brand etc) called predictor. This is known as regression.

- Classifier can sometimes work as regressor and vice-versa. for eg: Logistic regression can be used to find probability of a given class.
  - k-nearest Neighbour.
  - Linear regression
  - Logistic regression
  - Support Vector Machine.
  - Decision trees / Random forest
  - Neural Network.

- Unsupervised. - unlabelled data.

  - Clustering.
    - K-means
    - DBSCAN
    - Hierarchial Clustering Analysis (HCA)

  - Anomaly detection. and novelty detection.
    - One-class SVM
    - Isolation forest.

* Some NN architectures can be unsupervised, such as auto-encoders. and restricted Boltzmann machines. or semi-supervised, such in deep belief network and unsupervised pretraining..

  - Visualization or dimensionality reduction.
    - Principal Component Analysis
    - Kernel PCA.
    - Locally-linear Embedding (LLE)
    - t-distributed stochastic neighbour Embedding (t-SNE).

- Association rule learning.
  - Apriori
  - Eclat.

- In dimensionality reduction, in which the goal is to simplify the data without losing too much information.
  like age and mileage of a car may be correlated, so the DR algo. merge them into one feature that represent the car wears and tears.
  This is called feature reduction.

(c) Another imp unsuper task is anomaly detection.
For eg:- detecting unusual credit card transaction, catching defects etc.

(d) Novelty detection. They only expect to see normal data during training, while anomaly detection algo can withstand small percentage of outliners.

(e) Association rule learning:
try to dig large amount of data & discover interesting relation b/w diff xent attribute.

* Semi-supervised learning
eg:- google photo.

→ deep belief network:-
have unsupervised component called restricted boltzmann machine stacked on each other.
and then the system is fined-tuned using supervised training method.

* Reinforced learning
↳ Agent → obs environment
↳ rewards:- for steps
↳ learn policy.

Batch - learning

can't take new data and learn from incrementally
It has to train on all the data and from scratch
It is also called offline learning.

Online - learning.

You can train the system incretmently by feeding it data instances sequentially, either individual or by small group called mini - batch.

→ bad data can easily disturb the data
→ Use anomally detection.

* Instance Based → By heart learning
* Model - based learning → To model of these example and to make prediction.

* Challenges of ML

* insufficient Quantity of Training data.
* Non - representative Training data/ baised - data.
* Poor Quality data.
& Irrevelant Feature
  * Feature Selection :- Selecting most useful feature
  * Feature extraction :- Combining existing to produce more useful like dimension reduction.

Overfitting

It happen when the model is too complex relation to the amount and noisiness of training data.

* To simplify the model
* To gather more training data
* To reduce the noise in the training data.

* Regularization is used in linear regression model to make model more generally, by like keeping one parameter bounded

* A hyperparameter is the parameter for it might the parameter high the boundaries.

Underfitting.

↳ Simpler model for complex problem.

main soln.

* Selecting more complex model
* Feeding better feature (feature engineering)
* Reducing the constraint on the model.

* Testing & Validation.

Divide the data into training data & testing data

training error :- error in training data.

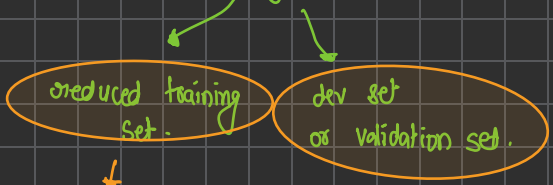
generalization error :- " " testing data.

TE ↓ but GE ↑ ⇒ Overfitting.

* Hyperparameter tuning and model selection.

* Choose model → Choose hyperparameter → Test.

training set. → for training the best model.



for training model

↳ For choosing best model.
and then use testing data/set for checking generalize.

* dev set small :⇒ evaluation impressive
dev set large : ⇒ It's like selecting best sprinter for a marathon.
one set ⇒ Cross Validation
many small Validation sets.

and averaging out for accurate measure.
Drawback, higher training time.

* Data mismatch.

* validation set & test must be representative as possible.
for eg 5k for test
5k for Validation.

* But, Now if model inform bad on devset, you want be able to say if the data overfitting or it is due to the mismatch b/w pic-from Web & app.

* Now for 3rd you make another so it called train dev set and set then to check for overfitting

End-to-End Machine Learning.

1. Frame the problem & look at big picture.
2. Get the data.
3. Explore the data to get insights.
4. Prepare the data to better expose the underlying data pattern to machine learning algorithm
5. Explore many different model and shortlist the best one.
6. Fine-tune your model and combine into great soln.
7. Present your soln.
8. Launch, monitor, and maintain your system.

* Pipeline.

A sequence of data processing component is called a data pipeline pipeline are very common as there are many data manipulation & dot transformation to do.

Each component is self-contained.
Component ⇒ take data from data store and spits out it (pipeline) data.

* Performance Measure

RMSE (Root mean square Error)

$$RMSE(X,h) = \sqrt{\frac{1}{m}\sum \left(h(x^{(i)}) - y^{(i)}\right)^2}$$

MAE (mean Absolute error)

$$MAE(X,h) = \frac{1}{m}\sum_{i=1}^{m} \left|h(x^{(i)}) - y^{(i)}\right|$$

→ system prediction function (hypothesis).

* RMSE ⇒ Euclidean norm $(\|\cdot\|)$ / ℓ_2 norm

* MAE ⇒ manhattan norm $(\|\cdot\|_1)$

* l_k norm $= \|v\|_k = (|v_0|^k + |v_1|^k + \dots + |v_n|^k)^{1/k}$

l_0 = no of non-zero element.

l_∞ = max abs value on the vector.

* higher the norm index, more focus on large value.

that's why RMSE is more sensitive than MAE
in case the are more outlier
thus MAE prefferred

* in case of bell shaped curve of outlier
thus RMSE is prefferred.

Here is the function to fetch the data:[11]

```
import os
import tarfile
import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

A type of compressing

Now when you call fetch_housing_data(), it creates a *datasets/housing* directory in your workspace, downloads the *housing.tgz* file, and extracts the *housing.csv* file from

(->) pd.read_csv( ) → to convert csv to DataFrame Class.

(->) pd.read_json( ) " " json " " "

<->DF.head() → Top 5 rows. & header.

.info() → For description.

.describe() → summary of numerical attributes

```
In [6]: housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude           20640 non-null float64
latitude            20640 non-null float64
housing_median_age  20640 non-null float64
total_rooms         20640 non-null float64
total_bedrooms      20433 non-null float64
population          20640 non-null float64
households          20640 non-null float64
median_income       20640 non-null float64
median_house_value  20640 non-null float64
ocean_proximity     20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

```
In [8]: housing.describe()
Out[8]:
```

	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20641.000000	20640.000000	20433.000(
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.38507(
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.00000(
50%	-118.490000	34.260000	29.000000	2127.000000	435.00000(
75%	-118.010000	37.710000	37.000000	3148.000000	647.00000(
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000(

```
%matplotlib inline  # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

↓ Size of one bar → Size (w x h)

* Creating a test set.

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

randomly chooses 20% and set them aside but can have generalization issue called data snooping bais train set might have more of one category and model might not get good contribution of all type of data.

↓

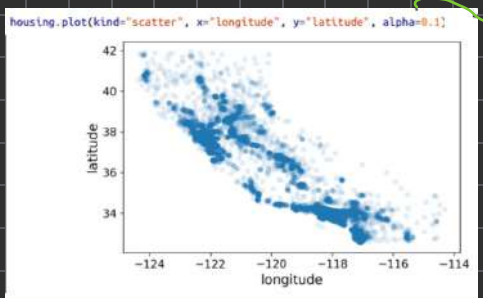
And not reproducible will always produce diffrent data set.

(1) We can use `np.random.seed(42)` to ensure reproduction on same data set but if we add new data, it will not have the same data in train & test set.

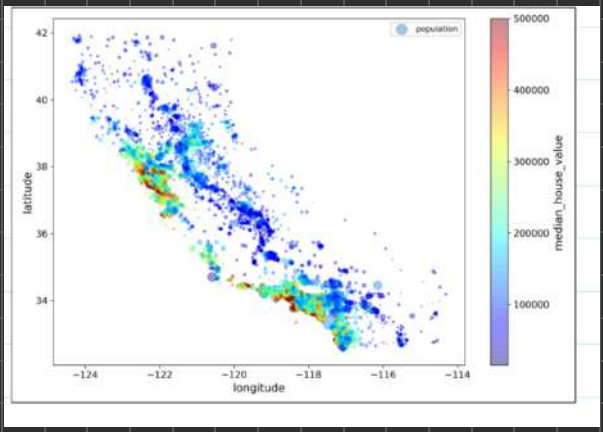
To solve the problem of same data not going the sets, we can use Hashing and whose hash is lower or equal to 20% of max value will be in test, it will assure that adding new data will not disturb the distribution of test set.

* Data Visualization & Insights.

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

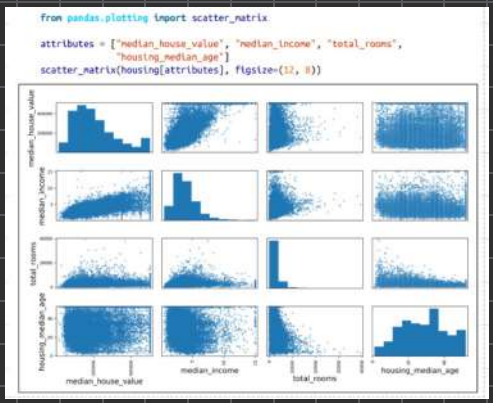


represent colour density.



* Looking for correlation.
  feature must be more correlated to the final output feature, but less correlated to each other other.

DF. corr() :→ Standard correlation coefficient



* Data Cleaning.
  for missing feature can.
  → Get rid of distinct or the row.
  → Get rid of whole feature
  → set the value to some value (0, mean, media).

* Design of Scikit - learn.

- Estimator
  * That can estimate some parameter. like imputer
    fit() ⇒ form estimator
    takes in DF.
  * any other parameter which it uses like it's strategy called as hyperparameters, can be changed by construction variable

- Transformer.
  * Some estimator (like "imputer") can make change in the DF itself.
    transform () ⇒ transform the DF.
  * Some have ⇒ fit - transform () ⇒ equivalent of fit (), transform (), but more optimized.

- Predictors.
  * Some estimator ("like linear regression) can make prediction
    - predict () → take DF of input and given DF of output or prediction.
    - Score () → measure quality, given a test set & its label.

* Inspection
  estimator's hyper parameters are accessible via public instance (e) imputer. strategy variable
  & learned parameter via, public instance variable with like imputer. statistics.

Handling text & Categorical Attributes.

ML need no.
So convert Convert Categorical Attributes to no.

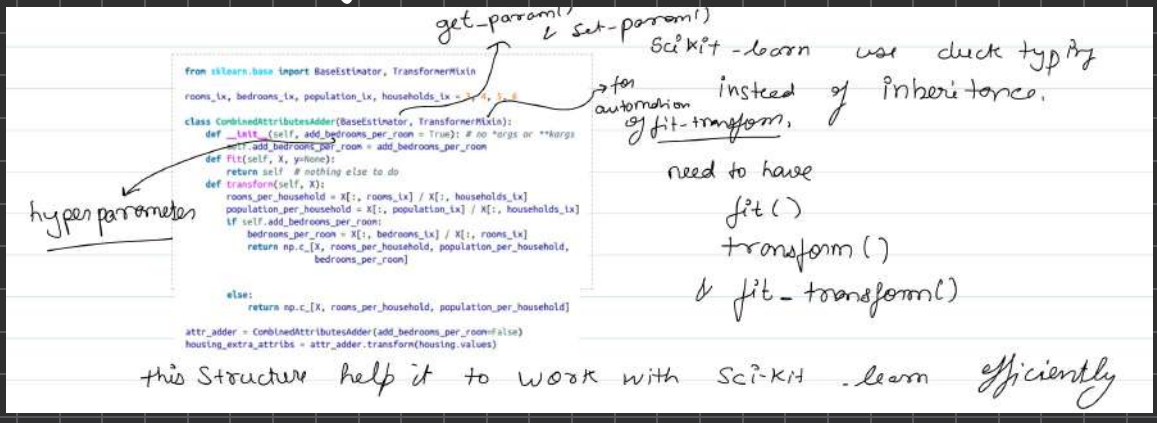
```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])

>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
      dtype=object)]
```

problem with Ordinal Encoder like model tends to relate near no like 2 & 3 to be more closely related but. that's not case here might be in series like this

['worst', 'worse', normal' good']

We need HOT Encoding → it makes no. of feature equal to no of different categories

* Custom Transformers.



Feature Scaling.

- min-max Scaling → make them b/w 0 & 1 ⇒ much effected by outline.

- Standardization → more preferred for gaussian type data.
  - StandardScaler(). make mean = 0 & σ = 1,
  - minMaxScaler() → have feature_range (hyperparameter) to change 0-1 or 0-10
    - transformer

* Transformation pipelines. → independent combination of data manipulation.

* we have handled categorical column, & numeric column separately.

* if we want to have 1 pipeline for whole data.

* Column transformer.

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

this need a (name, what to do, whom to do)

if we use drop in wt to do

if we use passthrough or just passes.

Note that the `OneHotEncoder` returns a sparse matrix, while the `num_pipeline` returns a dense matrix. When there is such a mix of sparse and dense matrices, the `ColumnTransformer` estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, `sparse_threshold=0.3`). In this example, it returns a dense matrix. And that's it! We have a preprocessing pipeline that takes the full housing data and applies the appropriate transformations to each column.

-> Fine - tune your model.

-> Grid Search

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

If `GridSearchCV` is initialized with `refit=True` (which is the default), then once it finds the best estimator using cross-validation, it retrains it on the whole training set. This is usually a good idea, since feeding it more data will likely improve its performance.

There are also more powerful imputers available in the `sklearn.impute` package (both for numerical features only):

- `KNNImputer` replaces each missing value with the mean of the *k*-nearest neighbors' values for that feature. The distance is based on all the available features.
- `IterativeImputer` trains a regression model per feature to predict the missing values based on all the other available features. It then trains the model again on the updated data, and repeats the process several times, improving the models and the replacement values at each iteration.

1. When a feature's distribution has a *heavy tail* (i.e., when values far from the mean are not exponentially rare), both min-max scaling and standardization will squash most values into a small range. Machine learning models generally don't like this at all, as you will see in Chapter 4. So *before* you scale the feature, you should first transform it to shrink the heavy tail, and if possible to make the distribution roughly symmetrical. For example, a common way to do this for positive features with a heavy tail to the right is to replace the feature with its square root (or raise the feature to a power between 0 and 1). If the feature has a really long and heavy tail, such as a *power law distribution*, then replacing the feature with its logarithm may help. For example, the `population` feature roughly follows a power law: districts with 10,000 inhabitants are only 10 times less frequent than districts with 1,000 inhabitants, not exponentially less frequent. Figure 2-17 shows how much better this feature looks when you compute its log: it's very close to a Gaussian distribution (i.e., bell-shaped).

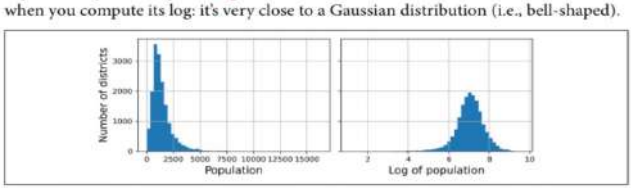


Figure 2-17. Transforming a feature to make it closer to a Gaussian distribution

2.

Another approach to handle heavy-tailed features consists in *bucketizing* the feature. This means chopping its distribution into roughly equal-sized buckets, and replacing each feature value with the index of the bucket it belongs to, much like we did to create the `income_cat` feature (although we only used it for stratified sampling). For example, you could replace each value with its percentile. Bucketizing with equal-sized buckets results in a feature with an almost uniform distribution, so there's no need for further scaling, or you can just divide by the number of buckets to force the values to the 0–1 range.

3.

When a feature has a multimodal distribution (i.e., with two or more clear peaks, called *modes*), such as the `housing_median_age` feature, it can also be helpful to bucketize it, but this time treating the bucket IDs as categories, rather than as numerical values. This means that the bucket indices must be encoded, for example using a `OneHotEncoder` (so you usually don't want to use too many buckets). This approach will allow the regression model to more easily learn different rules for different ranges of this feature value. For example, perhaps houses built around 35 years ago have a peculiar style that fell out of fashion, and therefore they're cheaper than their age alone would suggest.

4.

Another approach to transforming multimodal distributions is to add a feature for each of the modes (at least the main ones), representing the similarity between the housing median age and that particular mode. The similarity measure is typically computed using a *radial basis function* (RBF)—any function that depends only on the distance between the input value and a fixed point. The most commonly used RBF is the Gaussian RBF, whose output value decays exponentially as the input value moves away from the fixed point. For example, the Gaussian RBF similarity between the housing age x and 35 is given by the equation $\exp(-\gamma(x - 35)^2)$. The hyperparameter γ (gamma) determines how quickly the similarity measure decays as x moves away from 35. Using Scikit-Learn's `rbf_kernel()` function, you can create a new Gaussian RBF feature measuring the similarity between the housing median age and 35:

```
from sklearn.metrics.pairwise import rbf_kernel

age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]], gamma=0.1)
```

Figure 2-18 shows this new feature as a function of the housing median age (solid line). It also shows what the feature would look like if you used a smaller `gamma` value. As the chart shows, the new age similarity feature peaks at 35, right around the spike in the housing median age distribution: if this particular age group is well correlated with lower prices, there's a good chance that this new feature will help.

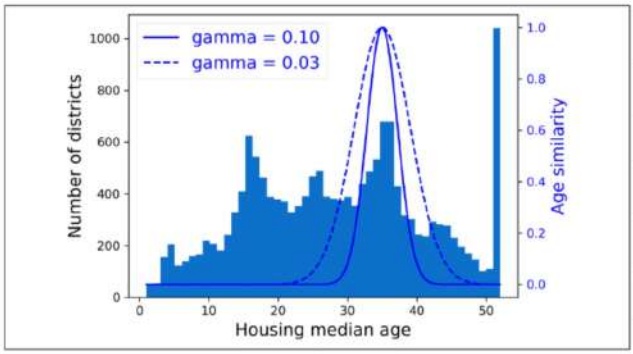


Figure 2-18. Gaussian RBF feature measuring the similarity between the housing median age and 35

So far we've only looked at the input features, but the target values may also need to be transformed. For example, if the target distribution has a heavy tail, you may choose to replace the target with its logarithm. But if you do, the regression model will now predict the *log* of the median house value, not the median house value itself. You will need to compute the exponential of the model's prediction if you want the predicted median house value.

Luckily, most of Scikit-Learn's transformers have an `inverse_transform()` method, making it easy to compute the inverse of their transformations. For example, the following code example shows how to scale the labels using a `StandardScaler` (just like we did for inputs), then train a simple linear regression model on the resulting scaled labels and use it to make predictions on some new data, which we transform back to the original scale using the trained scaler's `inverse_transform()` method. Note that we convert the labels from a Pandas Series to a DataFrame, since the `StandardScaler` expects 2D inputs. Also, in this example we just train the model on a single raw input feature (median income), for simplicity:

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5]  # pretend this is new data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

This works fine, but a simpler option is to use a `TransformedTargetRegressor`. We just need to construct it, giving it the regression model and the label transformer, then fit it on the training set, using the original unscaled labels. It will automatically use the transformer to scale the labels and train the regression model on the resulting scaled labels, just like we did previously. Then, when we want to make a prediction, it will call the regression model's `predict()` method and use the scaler's `inverse_transform()` method to produce the prediction:

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                   transformer=StandardScaler())
model.fit(housing[["median_income"]], housing_labels)
predictions = model.predict(some_new_data)
```

* We use cost function because.

- it's way to evaluate the best fit line
- it's way to measure performance how well your data performs.

$$J(m,b) = \frac{1}{2m} \sum_{i=0}^{m} (\hat{y}_i - y_i)^2$$

↓

Cost function

↳ 2 is due convention

CLASSIFICATION

MNIST → Dataset → most basic database with handwritten number.

DESCR → Description of dataset.

* Binary classifier
  * First ⇒ SGD Classifier
    Stochastic Gradient Descent.

  * It can handle very large dataset easily as it interact with training instances independently.

    * Suited for online learning

* Confusion matrix.

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53057,  1522],
       [ 1325,  4096]])
```

no of non-5 detected as 5 → False Positive

no of non-5 detected as non-5 → True Negative

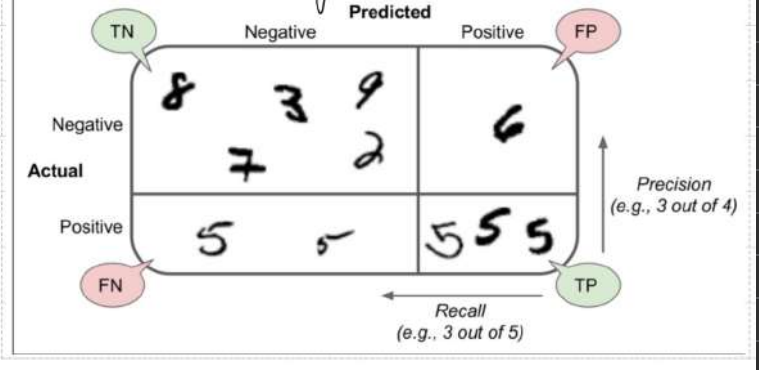
no of 5 detected as non-5 → False Negative

no of 5 detected as 5 → True positive

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

also called sensitivity.



•) F_1 source :- H.M of Recall & precision $= \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{Precision}}}$

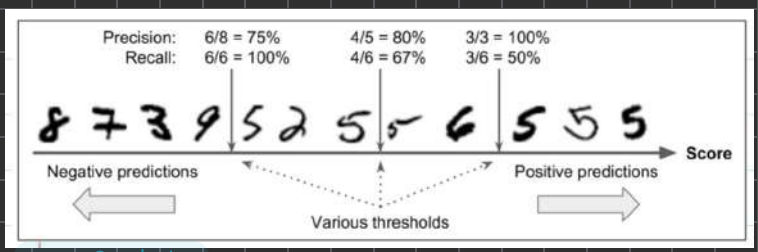
For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters in surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught).

* Precision / recall trade-off.

the model gives score based on a true to each prediction and set it's threshold.

now, if we inc thrs, we inc precision but dec recall.

•) to get the score and then we can customly set threshold.



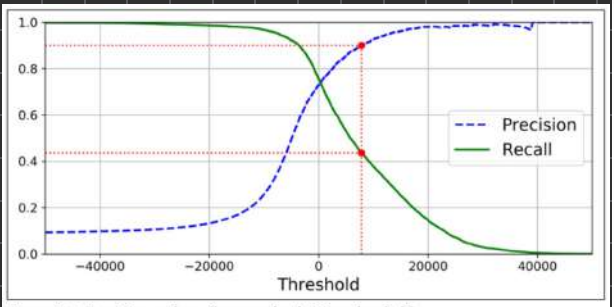


Figure 3-4. Precision and recall versus the decision threshold

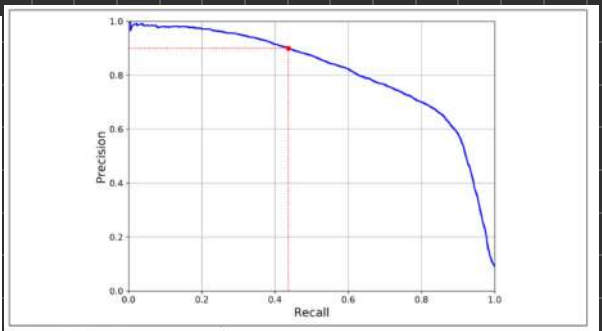


Figure 3-5. Precision versus recall

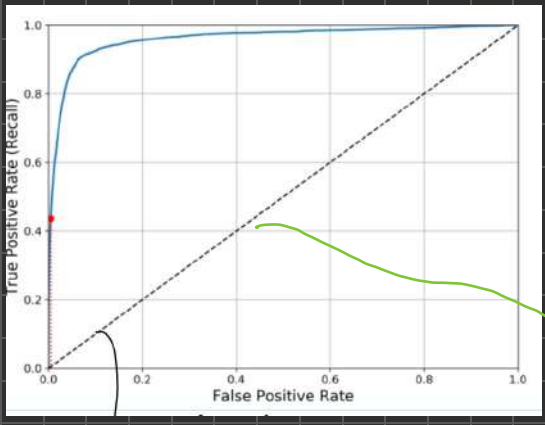
If someone says, "Let's reach 99% precision," you should ask, "At what recall?"

* ROC Curve.

Plots TPR + FPR

(also called Specificity)

→ gets fpr, tpr at various thresholds.



higher recall

⇓

higher (FPR).

to compare classifier we can also use ROC, under the curve.

total random predictor.

It should be near 1 for good classifier and 0.5 for random

* PR when we have low positives in and ROC AUC otherwise.

Since the ROC curve is so similar to the precision/recall (PR) curve, you may wonder how to decide which one to use. As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives. Otherwise, use the ROC curve. For example, looking at the previous ROC curve (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s). In contrast, the PR curve makes it clear that the classifier has room for improvement (the curve could be closer to the top-left corner).

* Multi class Classification.

* SGD, Random forest, naives Bayes classifier can handle multiple class natively.

* other's like.
  logistic & SVM classifier are strictly binary classifier.

* But we have many method to use many binary classifier to form multi-class classifier.

(a) OvR strategy ⇒ make binary classifier for all classes and select class with highest score.
↓
(one-versus the Rest)

(b) (OvO) strategy to form binary classifier. for each pair of class forming net classifier. $\left[\frac{n(n-1)}{2}\right]$

* it must be trained only on part of dataset which is it gonna distinguish.
  For eg 2 vs 3 classifier must be trained only on combined pics of 2 and 3.

* For some algo (like SVM), which scale poorly with size of dataset OvO is used. otherwise OvR is preffered.

* For some algo (like SVM), which scale poorly with size of data set OvO is used.

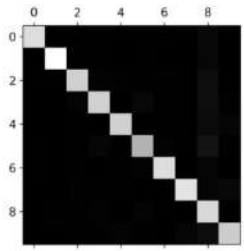
* Classes - atribute has the list of respective classes to the decision - function.

→ Error Analysis

→ Confusion matrix.

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



Now to focus on error
first find ratio of prediction by total no. of images in class
then fill diagonal with 0 as it is correct.
then we got error matrix.

* Now an ML engineer would try to reduce false 8. So for that it might create more training data for classes which are identified as but 8 but are not.

* Create new feature like loop or write an algo to count loops.

* Preprocessing with openCV to make loop or other feature stand out more.

* Multi output Classification.

Generalisation of multiclass classification, where each label can be multiclass.

TRAINING MODELS.

* linear regression model
  - Can be trained by using a direct "closed-form" equation to directly calculate it.
  - or using iterative optimization method called gradient descent. that gradually tweaks the model parameters to minimize the Cost function over the training and converge to the same soln as 1st one.

* Polynomial regression.
  - To fit non-linear data set.
  - How to avoid overfitting.

* Logistic regression
* Softmax "

Linear Regression.

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \cdots \theta_n x_n$$

$\hat{y}$ is predicated value
$n \Rightarrow$ no. of feature
$x_i \Rightarrow$ the i^{th} feature value
$\theta_j \Rightarrow$ the j^{th} model parameter
$\theta_0 \Rightarrow$ bais term
$\theta_1 + \theta_2 \cdots \theta_n \Rightarrow$ feature weights.

$$\hat{y} = h_\theta(x) = \theta \cdot x$$

θ = model parameter vector ($\theta_0 \Rightarrow$ bais term, $\theta_1 \cdots \theta_n$ the feature weights)
$x \Rightarrow$ feature vector (with x_0 to x_n, $x_0 = 1$)
$h_\theta \Rightarrow$ hypothesis function.

$$\hat{y} = \theta^T x.$$

To train minimize the performance measure (eg. RMSE) or MSE.
For eg let these regression hypothesis h_θ & the training set X.

$$MSE(X, h_\theta) = \frac{1}{m_i} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

The normal eqn.

$$\hat{\theta} = (X^T X)^{-1} X^T y \Rightarrow \text{for solution.}$$

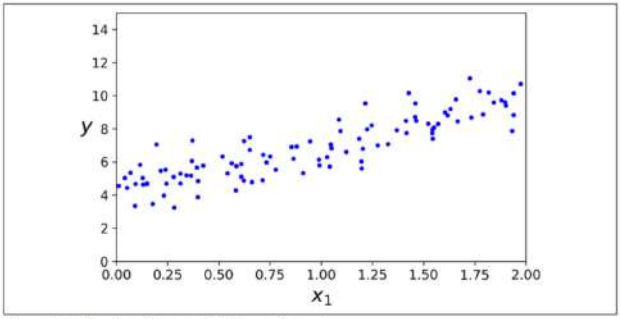
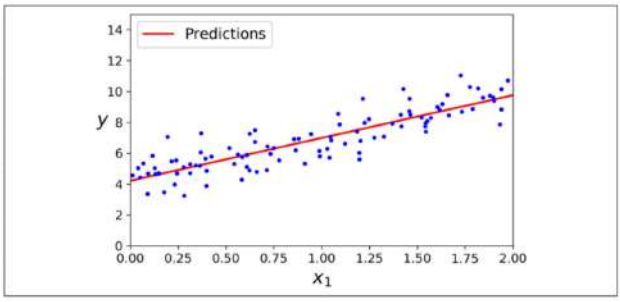


Figure 4-1. Randomly generated linear dataset

```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```



The linear regression class is based on scipy.linalg.lstsq().

It computes $\hat{\theta} = X^T y$, where X^T is pseudoinverse of X or moore inverse inverse

you can use np.linalg.pinv(X-b).dot(y).

* pseudo inverse is computed using a standard matrix factorization technique called singular value Decomposition. (SVD) that can decompose the X into matrix multiplication of three matrix

$$X^T = V \Sigma^T U^T.$$

* To compute Σ^T, algo take Σ set to zero all value below a tiny threshold value, and replaces all non-zero value with their inv and transfer the final results.

$$O(n^{2.4}) \text{ to } O(n^3).$$

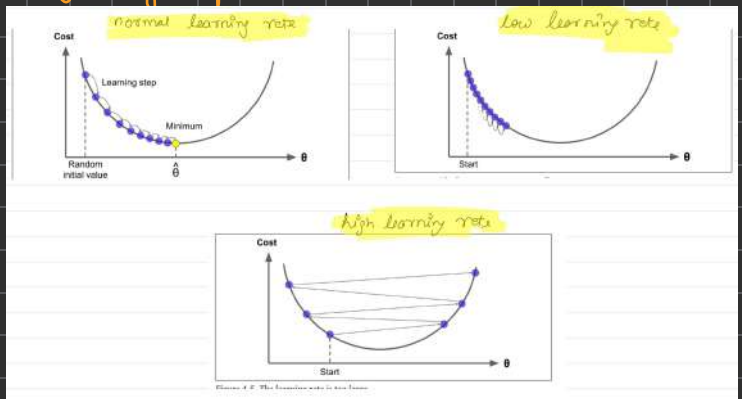
SVD
pseudo inverse :- $O(n^2)$. defined for all types of matrix.
but is $O(m)$ with relation that m is no of instances.

GRADIENT DESCENT.

idea :- Tweak parameter iteratively in order to minimize a cost function.

First :- random initialization.
then :- take step toward -ve slope.
then :- reach the minimum.

hyperparameter :- size of step.



problem :- not all cost function will be smooth and can have irregularities can can converge to local minimum.

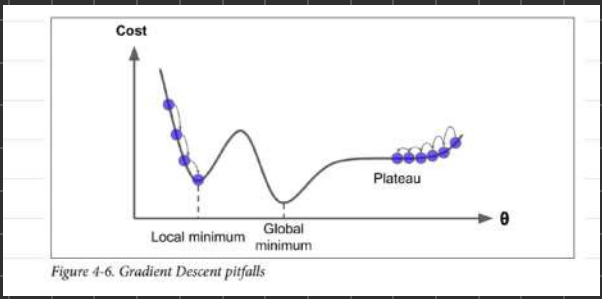


Figure 4-6. Gradient Descent pitfalls

But MSE :- Convex function and has only one minimum and smooth change in slope.
as its derivative is Lipschitz continous.

* Effect of scaling in gradient descent.

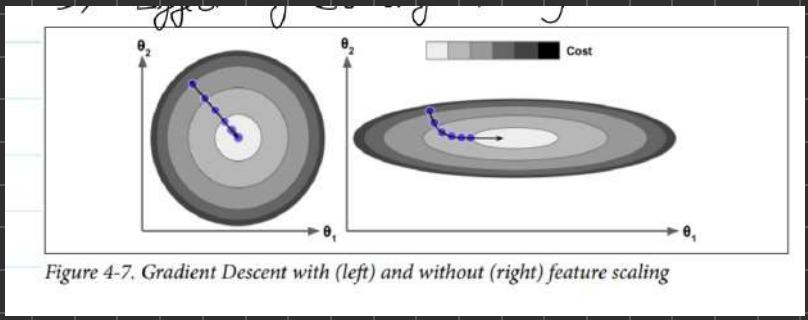


Figure 4-7. Gradient Descent with (left) and without (right) feature scaling

more parameter:- higher dimension → harder to locate minimum.

BATCH Gradient Descent.

$$\frac{\partial MSE(\theta)}{\partial \theta_j} = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)} \right) x_j^{(i)}$$

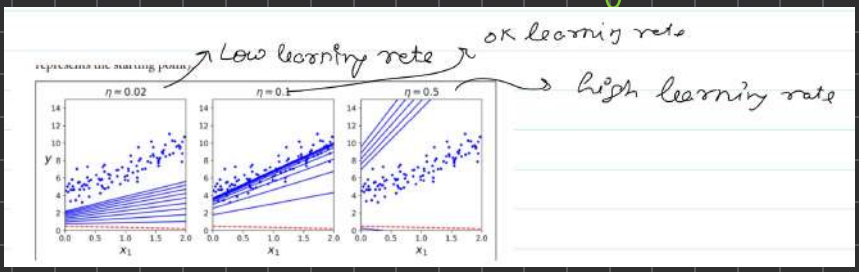
$$\nabla_\theta \cdot MSE(\theta) = \begin{pmatrix} \partial/\partial\theta_0 \; MSE(\theta) \\ \partial/\partial\theta_1 \; MSE(\theta) \\ \vdots \\ \partial/\partial\theta_n \; MSE(\theta) \end{pmatrix} = \frac{2}{m} X^T (X\theta - y)$$

In batch BGD, it calculate whole value of x at each step.

* Gradient descent step.

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta MSE(\theta)$$

η is learning step.



To find good learning rate, use grid search, but you might want to iterate limit the no of iteration, to eliminate slow model

Better way to add any tolerance (ε).

2.) Convergence rate

rate is $(1/\varepsilon)$

If you divide the ε by 10, you need to run the algo 10 times more.

Stochastic Gradient Descent.

In this, instead of calculating gradient descent on whole dataset it calculate on random instance on dataset.

(It can be used as out-of-core method, but it is randomized)

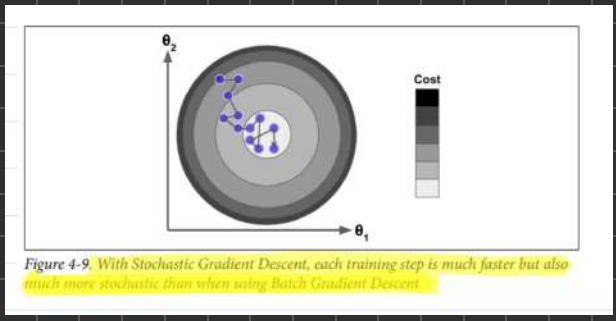


Figure 4-9. With Stochastic Gradient Descent, each training step is much faster but also much more stochastic than when using Batch Gradient Descent

once algo stops the parameters will be good but not optional.

* But it can actually jump out of local minima, has more chances of reaching global minima, instead of local minima.

* What we do is decrease the learning rate gradually, so. it can step become smaller & smaller & finally get settle on global minima.

* The function of learning rate is called learning schedule.

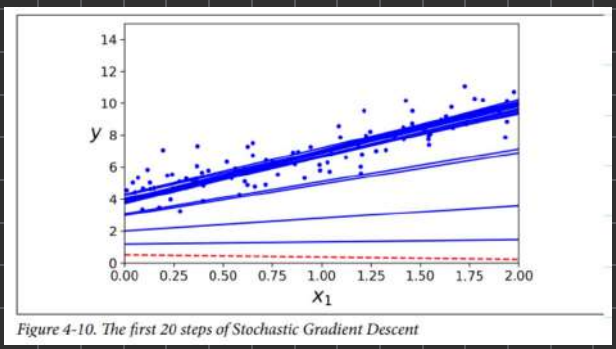


Figure 4-10. The first 20 steps of Stochastic Gradient Descent

* epoch is no of time the model goes through the dataset.

Batch DG goes through whole dataset 1000 times.

* Batch DG goes through whole dataset 10000 of time.

But SDG goes through less time, still reach a good sol?

Note that since instances are picked randomly, some instances may be picked several times per epoch, while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, another approach is to shuffle the training set (making sure to shuffle the input features and the labels jointly), then go through it instance by instance, then shuffle it again, and so on. However, this approach generally converges more slowly.

For SDG all training instance must be independent & identically distributed (IID) to ensure parameters are pulled toward global - maximum.

*) take random instances
*) shuffle the whole dataset.

* To perform linear regression using SGD use.

SGD Regressor class.

* do the training till 1000 epoch are done or 0.005 difference in loss is observed two adjoint epochs.

* Mini - batch gradient descent.
  -) uses random set of instance called as mini - batch to calculate gradient descent.
  -) you can get performance boost from hardcore optimization like by using GPU/ TPU.

* But its harder Batch GD for escape local minimas.

less erratic than Stochastic GD.

by creating mini - batch and using partial fit in batch - GD.

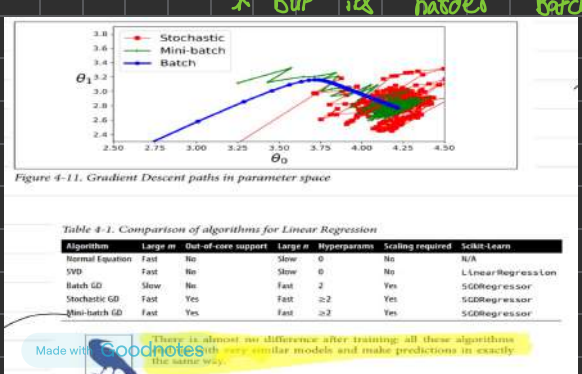


Figure 4-11. Gradient Descent paths in parameter space

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	LinearRegression
Batch GD	Slow	No	Fast	2	Yes	SGDRegressor
Stochastic GD	Fast	Yes	Fast	≥2	Yes	SGDRegressor
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	SGDRegressor

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

by creating mini batch and using partial fit in Batch GD

Polynomial Regression.

If data is more complex than a straight line.

* Add powers of each feature as a new feature and train a linear model
* This is polynomial Regression.

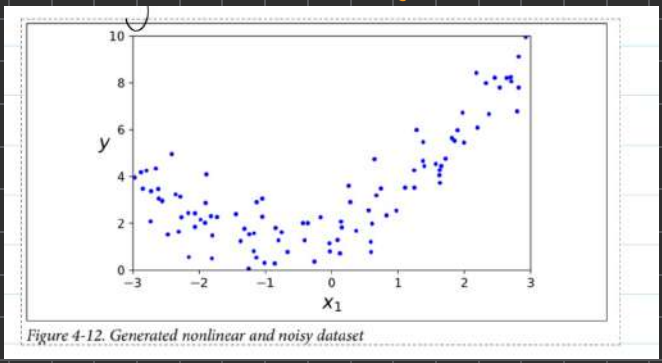


Figure 4-12. Generated nonlinear and noisy dataset

Clearly a straight line would never fit into this.

model approx = $\hat{y} = 0.56\,x_1^2 + 0.93x_1 + 1.73$

$\hat{y} = 0.5x_1^2 + 1x_1 + 2 + \text{gaussian noise}$

* In polynomial feature with deg = 3 would not only add - the feature a^2, a^3, b^2, b^3, but also ab, a^2b, ab^2.

* In polynomial feature (deg = d) transform an array containing n. feature into array containing $\frac{(n+d)!}{n!\,d!}$. Notional Expansion of the no. of feature.

Learning Curves.

* high-degree polynomial regression is like to fit the data. much more better than linear regression.

* But can over fit the data easily for eg:-

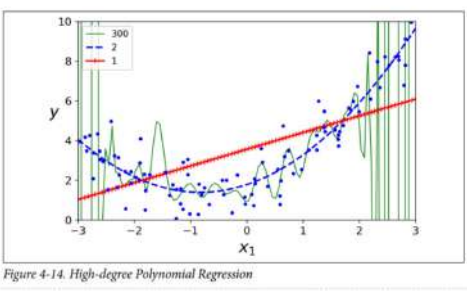


Figure 4-14. High-degree Polynomial Regression

300 degree polynomial overfit the data.

Another way is check using learning curves.

* Plots of the model's performance on the training set & validation set as a function of the training set size (or the training iteration).

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```

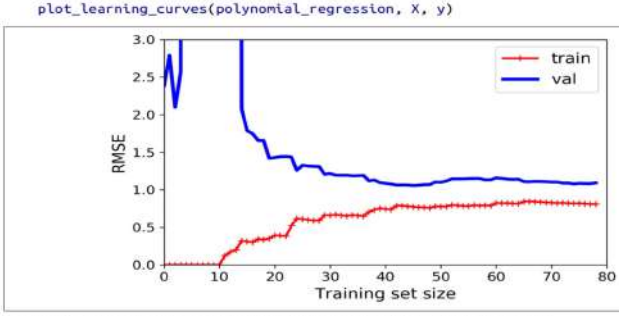


Figure 4-16. Learning curves for the 10th-degree polynomial model

this look almost same bet-

→ error is much lower than with the linear regression.

→ There is a gap b/w the curves This means the model overfit the data as it perform much better in training set.

* One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

BIAS / Variance Trade-off.

* Imp theoritical result of statics & machine learning is the fact that a model's generalization error is sum of.

(i) Bias (ii) Variance (iii) Irreducible error.

(i) Bias:- This part is due to wrong assumption, such that assumption that the data is linear when it actually quadratic. A high-bias will under fit the data.

(ii) Variance:- This part is due to model's excessive sensitivity to small variation in training data. A model with high degree of freedom is likely to high-variance & overfit the data (degree).

(iii) Irreducible error:- This part is due to the noisiness of the data itself to reduce this, fix the data sources, such as broken sensors, or detect and remove outliers.

REGULARIZED LINEAR MODEL.

↓

to reduce its degrees of freedom by reducing parameter and making them hyperparameter.

Methods → Reducing parameter
Regularization
Early Stopping.

* Ridge Regression (Tikhonov regression)
  - is regularized version of linear Regression, regularization term equal to $\alpha \sum_{i=1}^{n} \theta_i^2$ is added to cost function.
  - This forces the model to keep weights as small as possible. {regularization should be done only train not in validate}.

It is quite common for the cost function used during training to be different from the performance measure used for testing. Apart from regularization, another reason they might be different is that a good training cost function should have optimization-friendly derivatives, while the performance measure used for testing should be as close as possible to the final objective. For example, classifiers are often trained using a cost function such as the log loss (discussed in a moment) but evaluated using precision/recall.

$$J(\theta) = MSE(\theta) + \alpha \cdot \frac{1}{2} \sum_{i=1}^{n} \theta_i^2 \quad \{\theta_0 \text{ is not regularized}\}$$

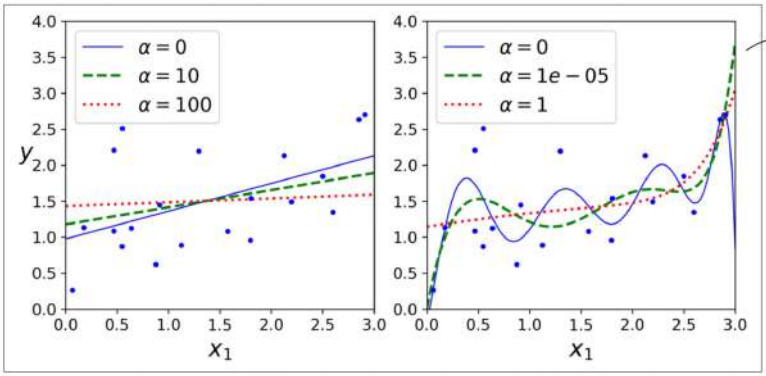
{Ridge Regression cost function}.

* If w is vector of feature weights (θ_1 to θ_n) then regularization terms $= \frac{1}{2}(\|w\|_2)^2$

For gradient descent.
add αw to MSE gradient vector.

* (use standard scaler) before performing ridge Regression as it sensitive to scale the input feature.

Ridge Regression ←



→ Polynomial Regression with Scaling & Ridge regularization

Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

Equation 4-9. Ridge Regression closed-form solution

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{A})^{-1}\ \mathbf{X}^T\ \mathbf{y}$$

It is (n+1)×(n+1) identity matrix with 0 in top-left cell, corresponding to bais term.

* Lasso Regression.

Least absolute Shrinkage and Selection operator Regression, also called as lasso regression.

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

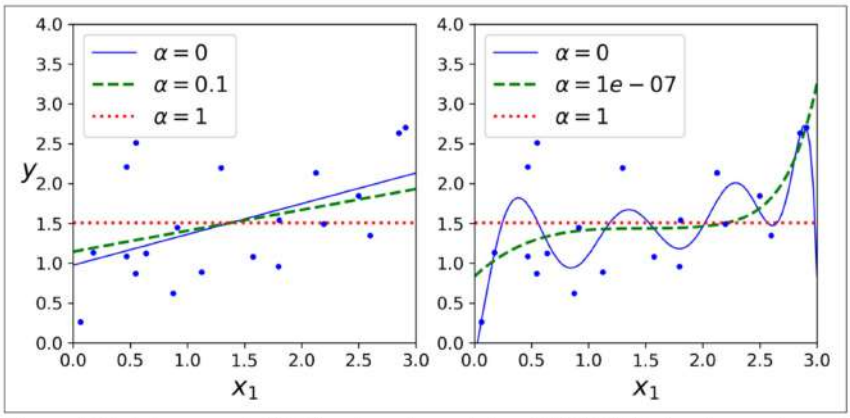
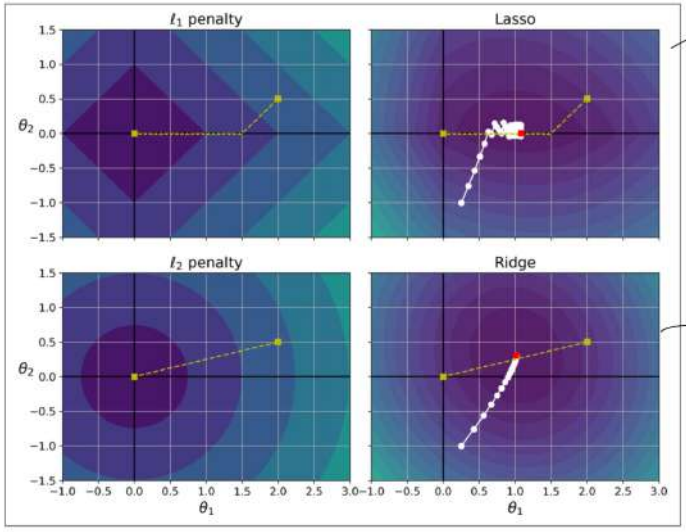


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

* Imp characteristic of lasso is that it tends to eliminate the weight of least imp feature (i.e set them to zero).

* Lasso automatically perform feature selection & output a sparse model (i.e with few non-zero feature weights).



→ Always wiggle around as derivative of $|x|$ is 1 or -1

→ derivative is $-x$ or $+x$

Figure 4-19. Lasso versus Ridge regularization

To avoid Gradient Descent from bouncing around the optimum at the end when using Lasso, you need to gradually reduce the learning rate during training (it will still bounce around the optimum, but the steps will get smaller and smaller, so it will converge).

We use subgradient function for lasso when any $\theta_i = 0$

Equation 4-11. Lasso Regression subgradient vector

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \operatorname{sign}(\theta_1) \\ \operatorname{sign}(\theta_2) \\ \vdots \\ \operatorname{sign}(\theta_n) \end{pmatrix} \quad \text{where } \operatorname{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

* Elastic Net.

middle ground b/w Lasso and Ridge with a control mix ratio r.

$r = 0 \Rightarrow$ EN = Ridge

$r = 1 \Rightarrow$ EN = Lasso.

Equation 4-12. Elastic Net cost function

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

* We shouldn't use linear regr.
* Ridge as default is good.
* If you suspect, some of feature are not useful

* EN / Lasso.

b/c lasso might behave erratically when the no. feature > no. of instance.

* Early Stopping

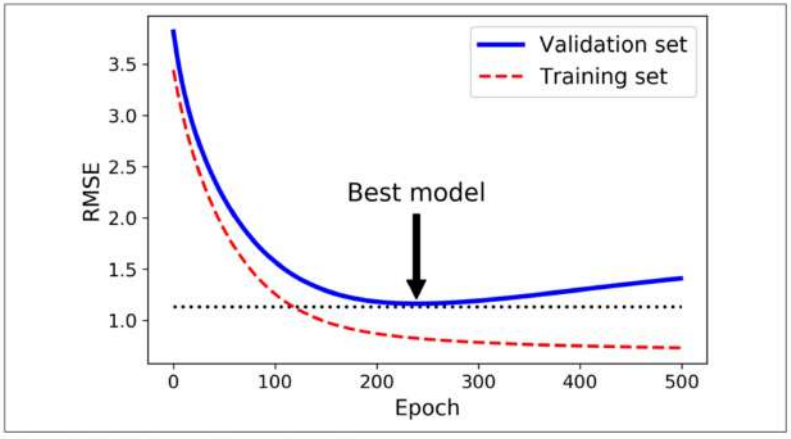


Figure 4-20. Early stopping regularization

* Stopping when error in validation step become. It so beautiful that geoffrey hinton called it "beautiful free lunch".

With Stochastic and Mini-batch Gradient Descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not. One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum.

Logistic Regression.

* Also known logit regr. is commonly used to estimate the probability that the instance belong to a particular class. If it is probability is >50 then yes

* Estimating probability,

$\hat{p} = h_\theta(x) = \sigma(x^T \theta)$

σ is sigmoid function (s-shaped) a no. b/w 0 & 1.

$$\sigma(t) = \frac{1}{1+\exp(-t)}.$$

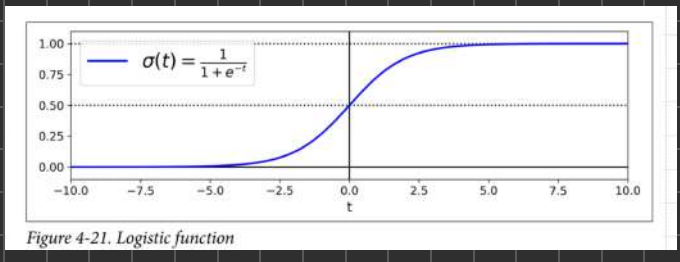


Figure 4-21. Logistic function

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.0 \end{cases}.$$

Logistic Regression model prediction.
Sigmoid is often called logit.

as $\text{logit}(\hat{p}) = \log\left(\frac{\hat{p}}{1-\hat{p}}\right)$

log-odds.

* Training & Cost function

idea ⇒ high probability for $(y=1)$ & low probabs for $(y=0)$

* Cost function of a single training instance.

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y=1 \\ -\log(1-\hat{p}) & y=0 \end{cases}$$

Equation 4-17. Logistic Regression cost function (log loss)

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(\hat{p}^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-\hat{p}^{(i)}\right)\right]$$

no close form

But cost function is convex, ⇒ so gradient descent is gaursanteed to find global minimum.

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma(\theta^T x^{(i)}) - y^{(i)}\right) x_j^{(i)}$$

The hyperparameter controlling the regularization strength of a Scikit-Learn `LogisticRegression` model is not `alpha` (as in other linear models), but its inverse: `C`. The higher the value of `C`, the *less* the model is regularized.

Softmax Regression.

The logistic regressor can be generalized to support multiple class directly, without having to train & combine multiple binary classifier. This is also called softmax regression as multinomial logistic regressor.

idea ⇒ Calculate $S_k(x)$ for each class & estimate the probability by applying the softmax function. (normalized exponential).

* Softmax score for class K.

$$S_k(\theta) = x^T \theta^{(k)}$$

↳ logit or log-odds.

* Softmax function

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

↓

having score for each class.

$$\hat{y} = \underset{k}{\arg\max}\ [\sigma(s(x))_k] = \underset{k}{\arg\max}\ S_k(x) = \underset{k}{\arg\max}\ ((\theta^{(k)})^T x)$$

(Can't be used for multi output)

Training ⇒ Cross entropy cost function

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

↳ target probability.

for 2 class it is equal to logistic regression cost func^n.

Cross Entropy

Cross entropy originated from information theory. Suppose you want to efficiently transmit information about the weather every day. If there are eight options (sunny, rainy, etc.), you could encode each option using three bits because $2^3 = 8$. However, if you think it will be sunny almost every day, it would be much more efficient to code "sunny" on just one bit (0) and the other seven options on four bits (starting with a 1). Cross entropy measures the average number of bits you actually send per option. If your assumption about the weather is perfect, cross entropy will be equal to the entropy of the weather itself (i.e., its intrinsic unpredictability). But if your assumptions are wrong (e.g., if it rains often), cross entropy will be greater by an amount called the *Kullback–Leibler (KL) divergence*.

The cross entropy between two probability distributions p and q is defined as $H(p,q) = -\Sigma_x p(x) \log q(x)$ (at least when the distributions are discrete). For more details, check out my video on the subject.

* gradient vector

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)} \right) x^{(i)}$$

Decision Boundaries.

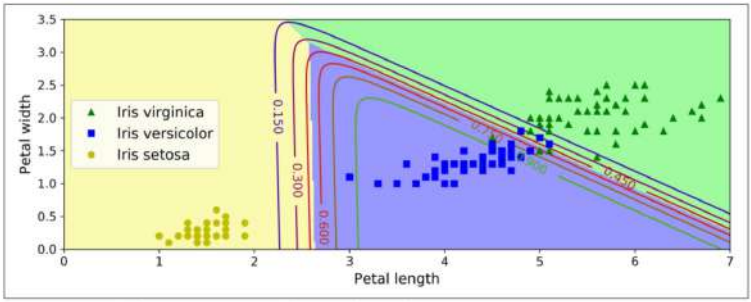


Figure 4-25. Softmax Regression decision boundaries

SUPPORT VECTOR MACHINE

* Can perform linear or non-linear classification, regression and even outlier detections.

* Suited for small & medium size data set.

* Large margin classification.
  Forming a decision boundary with equal distance from both boundaries.

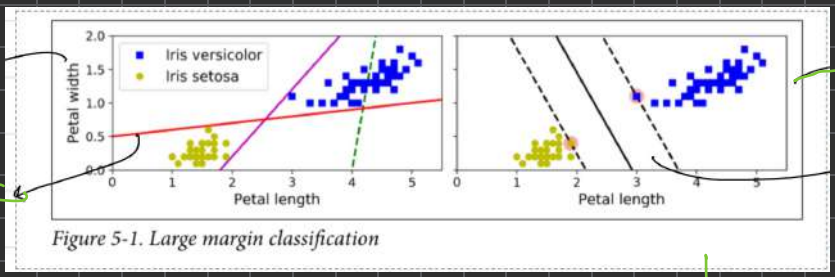


Figure 5-1. Large margin classification

3 possible classification

Large margin classification

↳ It forms like a street.

→ Adding more data "off the street" will have no effect. on the decision boundary. and are only effected by instance on the edge of street. These instance are called Support Vector.

* SVM are sensitive to scaling.

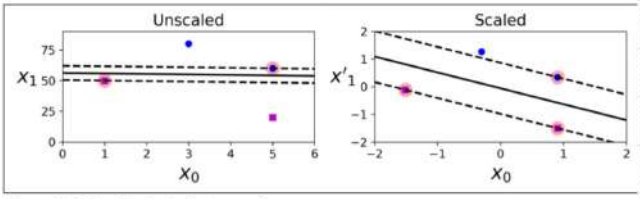


Figure 5-2. Sensitivity to feature scales

SVMs are sensitive to the feature scales, as you can see in Figure 5-2: in the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using Scikit-Learn's StandardScaler), the decision boundary in the right plot looks much better.

* Soft Margin classification.

If we say all instance must off the street and on one side then it is called hard - margin classification.

Cons:- -> Only work for linearly separable data.
-> highly effected by outlier.

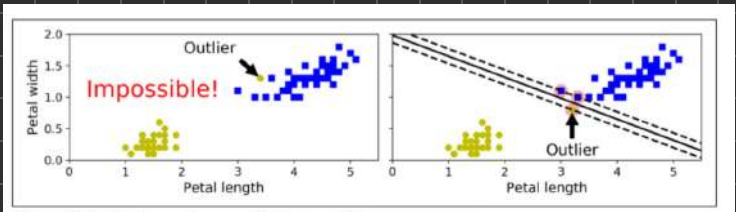


Figure 5-3. Hard margin sensitivity to outliers

To avoid this we allow some violations and then classification is called soft margin.

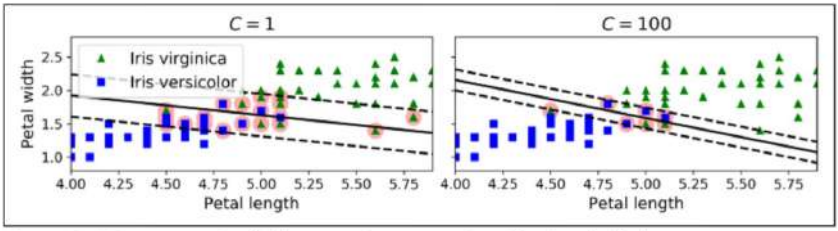


Figure 5-4. Large margin (left) versus fewer margin violations (right)

higher C hyperparameter represents - low no. of margin violations.

* If SVM is overfitting reduce C.

* Unlike logistic regression classifier, SVM classifier do not output probabilities for each each.

```
From sklearn.svm import SVC

SVC(kernel = "linear", C=1).
```

or From Sklearn . linear model import.

SGD Classifier (loss = "hinge" , alpha = $\frac{1}{m*c}$)

It apply Stochastic gradient descent.

It is slow but can handle online classification.
or huge dataset. (out - of - core training
And for better performance set dual hyperparameter to false.

* Non - linear SVM classification.

some data sets are not even close to linear.

* Add polynomial feature.

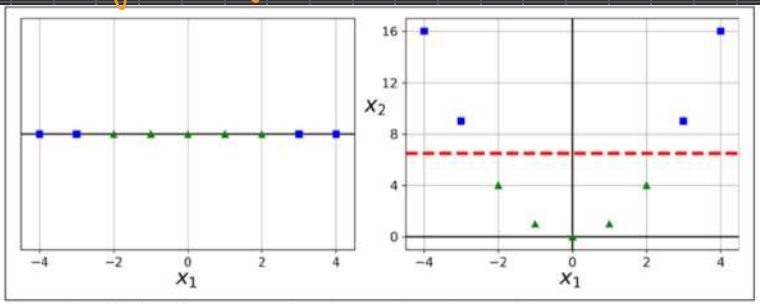


Figure 5-5. Adding features to make a dataset linearly separable

Another eg.

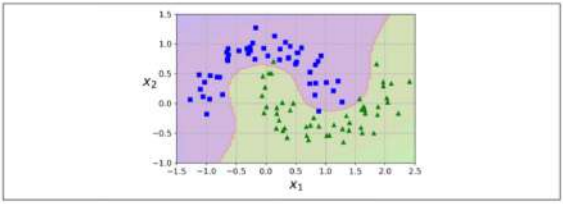


Figure 5-6. Linear SVM classifier using polynomial features

Cons :- At low polynomial degree, it cannot deal with very complex dataset.

* At high polynomial degree, it create huge number of feature

* Polynomial Kernel.

- to avoid this, we can use "kernel trick", it gives same result as adding polynomial features without adding them.

-

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

- Coeff 0 is parameter represent high much model is influency by high degree vs low degree polynomial.

If overfitting, dec degree ↓
If underfitting, inc degree

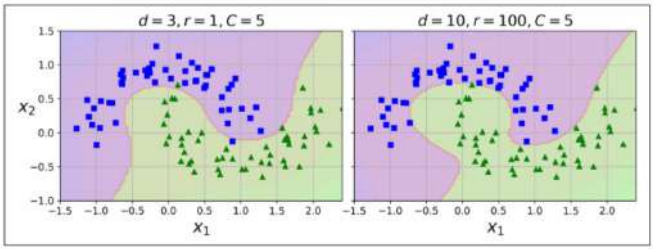
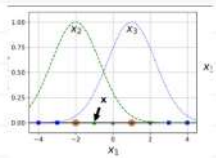


Figure 5-7. SVM classifiers with a polynomial kernel

* Common approach is to coarse grid-search and again a fine-grid search around the best values.

Similarity Features

* Another way to tackle non-linear problem is to add features using a similarity function.
* Which measures how each instance resemble a particular landmark.

For eg ⇒ in 

, we added two landmark at $x_1 = -2$ & $x_1 = 1$
and if we define similarity function to be Gaussian Radial Basis Function (RBF) with $\gamma = 0.3$

$$\phi_\gamma(x, \ell) = \exp(-\gamma \|x - \ell\|^2).$$

* A bell-shaped function varying from 0 to 1 at the landmark
Now, look at $x = 1$, it is 1 away from 1st landmark & from second.
Therefore it new feature ⇒ $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$
$\exp(-0.3\ x_2^2) \approx 0.26$
and now they become linearly seperable.

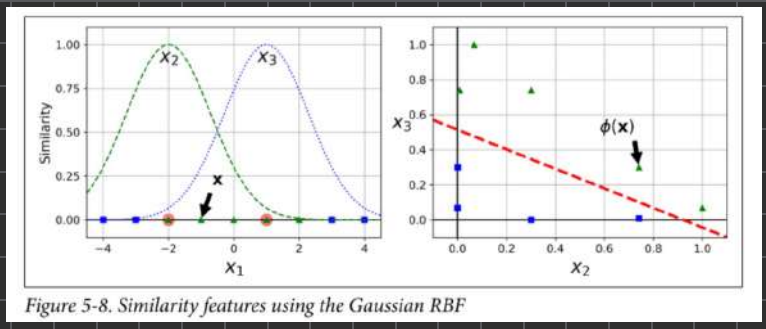


Figure 5-8. Similarity features using the Gaussian RBF

Q How to choose landmark?
Simple approach:- to create a landmark at each and every instance then increasing dimension & increasing the chance of becoming linearly separable.

Con:- training set with m instance & n feature will transform to m instance & m feature (given droping original feature).

* Gaussian RBF kernel.

Again kernel trick, without creating and getting similar result.

hyper parameter gamma & c.

↳ bell curve shape narrower and small range for each instance, thus irregular. around instances of decision boundary.

So overfitting reduce gamma
underfitting, increase it similar to c.

other kernel: Rare
one is String kernel ∃ used to classify text document or DNA sequence.

Qs How to decide which kernel to use.

<·> first ⇒ linear SVC ↝ (much faster than SVC (kernel = "linear"))
↓
if data set is large or too many instances.

<·> not too large ⇒ try gaussian RBF.

<·> Sparse time, try a grid search with other kernel and if there are special kernel for your training data structure.

* Computational Complexity

don't support kernel trick.

linear SVC Class: O(m×n), if you need high precision ε {"tol"} hyper parameter.

SVC Class :- Support kernel tricks based on libsvm ⇒ become dreadfully slow as we inc no. of instances.

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
LinearSVC	$O(m \times n)$	No	Yes	No
SGDClassifier	$O(m \times n)$	Yes	Yes	No
SVC	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM Regression.

Also support linear & non-linear regression.

(->) Wt is do, instead of keeping data off the track, it keeps them on the track, while limiting margin violation ("off the street")

(->) And epsilon is the hyperparameter, higher epsilon, higher the of the street.

* Adding more training data inside the street does not effect the model's prediction, => thus => ε- insensitivity.

=> Linear SVC <=> Linear SVR

=> SVC <=> SVR

UNDER THE WOOD

bais term = b

Weight Vector = W

$$\hat{y} = \begin{cases} 0 & W^T x + b < 0 \\ 1 & W^T x + b \geq 0 \end{cases}$$

* n-parameter => Decision function is n- dimensional hyperplane & decision boundary is (n-1) dimensional hyperplane

* Training linear SVM: is finding value of W b that make margins wide as possible while avoid margin violations. (hard margin)

or lining them (soft margin)

* Training Objective

- $\|w\|$ is slope of the function. If we divide it by 2, the points where decision function is ± 1 are going to twice as far away from the boundary thus increasing the boundary.

aim minimize $\|w\|$
and we want to avoid (minimize margin violator
If we define $t^{(i)} = -1$ for $y^{(i)} = 0$
& $t^{(i)} = 1$ for $y^{(i)} = 1$

** $t^{(i)}(w^T x^{(i)} + b) \geq 1$ → constraint become

* Hard margin linear SVM classifier objective
minimize $\frac{1}{2} w^T w$

Subject to $t^{(i)} (w^T x^{(i)} + b) \geq 1$ for $i = 1, 2, 3 \dots m$.

We minimize $\frac{1}{2} w^T w$ ≠ $\frac{1}{2} \|w\|^2$ as has a nice simple derivative which is w itself, while $\|w\|$ is not differentiable $w=0$

* To get the soft margin, we introduce slack function $z^{(i)} \geq 0$ for each instance (i). $z^{(i)}$ measure how much i^{th} instance is allowed to violate the margin.

* Now have two objective to make $\frac{1}{2} w^T w$ small & make slack variable smaller to reduce the margin violation (This is where C parameter help).

* Soft margin linear SVM classifier objective

$$\underset{w, b, z}{\text{minimize}} \quad \frac{1}{2} w^T w + C \sum_{i=1}^{m} z^{(i)}$$

Subject to $t^{(i)} (w^T x^{(i)} + b) \geq 1 - z^{(i)}$ and $z_i^{(i)} \geq 0$ for $i = 1, 2, 3 \ldots m$.

These are type of Quadratic programming problem.

General problem.

$$\text{minimize} \quad \frac{1}{2} p^T H p + f^T p.$$

$$\text{Subject to} \quad Ap \leq b.$$

where.
- p - is an n_p-dimensional vector (n_p = no. of parameters)
  - it is an $n_p \times n_p$ matrix
- f is an n_p-dimensional vector.
- A is an $n_c \times n_p$ matrix (n_c = no. of constraints).
- b is n_c-dimensional vector.

The Dual problem.

Given a constrained optimization problem, known as primal problem, it is possible to express a different but closely related problem, called it's dual problem. The solution for dual gives us lower bound to the solution of the primal problem but under some condition. It gives the same result. & SVM follows that condition.

{ Condition was:- the objective function is convex, and inequality constraints are continuously differentiable and convex function }.

* Dual form of the linear SVM objective

$$\underset{\alpha}{\text{minimize}} \quad \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}x^{(i)T}x^{(j)} - \sum_{i=1}^{m}\alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for} \quad i = 1, 2, \cdots m.$$

→ From dual solution to primal solution.

$$\hat{w} = \sum_{i=1}^{m}\alpha^{(i)} \cdot t^{(i)} \cdot x^{(i)}$$

$$\hat{b} = \frac{1}{n_s}\sum_{i=1}^{m}\left(t^{(i)} - \hat{w}^T x^{(i)}\right)$$

It is faster to solve when no of training instance are smaller than no. of problem

And the kernel trick is possible in dual problem, not in primal problem.

SVM Dual problem

* Lagrange multiplier :- To transform constrained optimization objective into an unconstrained one, by moving the constrained into objective function.

For eg!-

$f(x,y) = x^2+2y$

Const $= 3x+2y+1$

Lag rangain (or lagrange function):- $g(x, y, \alpha) = f(x,y) - \alpha(3x+2y+1)$

* Joseph- louis lagrange showed that if lagrange multiplier $(\hat{x}, \hat{y})$ in solution to the constrained optimization problem. there must exist $\hat{\alpha}$ such that $(\hat{x}, \hat{y}, \hat{\alpha})$ is a stationary point of lagrangian

meaning all partial derivative are zero

However this only apply only to equality constraints.

under some regularity condition (svm follows), this method can be generalized to inequality condition too.

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^{m} \alpha^{(i)} \left(t^{(i)} (w^T x^{(i)} + b) - 1\right)$$

with $\alpha^{(i)} \geq 0$ for all $i = 1, 2, \ldots m$.

* if there is a solution, it will be in stationary point $(\hat{w}, \hat{b}, \hat{\alpha})$. that respect KKT condition.

$t^{(i)} (w^T x^{(i)} + \hat{b}) \geq 1$ for all i

$\alpha^{(i)} \geq 0$

either $\alpha^{(i)} = 0$ or i^{th} constraint be on active constraint which is $t^{(i)}(\hat{w}^T x^{(i)} + b) = 1$. It is called complementary slackness condition. It implies that either $\hat{\alpha}^{(i)} = 0$ or i^{th} instance lies on the boundary. (It is a support vector).

Partial derivative.

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum \alpha^{(i)} \cdot t^{(i)} \cdot x^{(i)}$$

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = - \sum_{i=1}^{m} \alpha^{(i)} t^{(i)}$$

prop of stationary problem.

$$\hat{w} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} x^{(i)}$$

$$\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} = 0$$

putting it in.

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^{m} \alpha^{(i)} \left(t^{(i)}(w^T x^{(i)} + b) - 1\right)$$

$$= \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} x^{(i)T} x^{(j)} - \sum_{i=1}^{m} \alpha^{(i)}$$

with $\alpha^{(i)} \geq 0$

- Support Vector $= t^{(i)}(\hat{w}^T x^{(i)} + \hat{b}) = 1$

$$\hat{w}^T x^{(i)} + \hat{b} = t^{(i)}$$

$$\Rightarrow \hat{b} = t^{(i)} - w^T x^{(i)} \quad \pm 1$$

* Avg over all support Vector $\Rightarrow \hat{b} = \frac{1}{n_s} \sum_{i=1}^{m} [t^{(i)} - \hat{w}^T x^{(i)}]$

Kernelized SVM

$$\phi(x) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}\, x_1 x_2 \\ x_2^2 \end{pmatrix}$$

$$\phi(a)^T \phi(b) = \begin{pmatrix} a_1^2 \\ \sqrt{2}\, a_1 a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}\, b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2 a_1 b_1 a_2 b_2 + a_2^2 b_2^2$$

$$(a_1 b_1 + a_2 b_2)^2 = \left[\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right]^2$$

So $\phi(a)^T \phi(b) = (a^T b)^2$ $\qquad = (a^T b)^2$

- If you apply ϕ to all training instance, there will be $\phi(x^{(i)})^T \phi(x^{(j)})$

But if ϕ is 2nd degree polynomial transformation.
this can be replace by $(x^{(i)} x^{(j)})^2$.

A kernel function $K(a,b)$ is a function capable by computing $\phi(a)^T \phi(b)$ based only on a & b without knowing ϕ.

Linear $\Rightarrow$ $K(a,b) = a^T b$
Polynomial $\Rightarrow$ $K(a,b) = (\gamma a^T b + r)^d$
gaussian RBF $\Rightarrow$ $K(a,b) = \exp(-\gamma \|a-b\|^2)$.
Sigmoid $\Rightarrow$ $K(a,b) = \tanh(\gamma a^T b + r)$

To make prediction.

$$h_{\hat{w},\hat{b}}(\phi(x^{(n)})) = \hat{w}^T \phi(x^{(n)}) + \hat{b}.$$

$$\hat{w} = \sum_{i=1}^{m} \alpha^{(i)} \cdot t^{(i)} x^{(i)}$$

↳ general.

$$= \left(\sum_{i=1}^{m} \alpha^{(i)} \cdot t^{(i)} \cdot \phi(x^{(i)}) \right)^T \phi(x^{(n)}) + \hat{b}$$

$$= \sum \alpha^{(i)} t^{(i)} \phi^T(x^{(i)}) \phi(x^{(n)}) + \hat{b}$$

$$= \sum_{i=1}^{m} \hat{\alpha}^{(i)} \cdot t^{(i)} K(x^{(i)}, x^{(n)}) + \hat{b}.$$

$\alpha^{(i)} > 0$

Using Kernel trick for bais term.

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \alpha^{(i)} > 0}}^{m} \left(t^{(i)} - \sum \hat{\alpha}^{(i)} t^{(j)} K(x^{(i)}, x^{(j)}) \right)$$

Online SVMs.

1. learning incrementally.
2. gradient Descent (using SGD classifier)
3. much slow than QP

→ Linear SVM classifier cost function.

$$J(w, b) = \frac{1}{2} w^T w + C \sum_{i=1}^{m} \max(0, 1 - t^{(i)}(w^T x^{(i)} + b))$$

Decision Trees.

* Decision Tree base of Random forest.
* CART Training algorithm by scikit-learn.

* Training and visualize a decision tree.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

→ To train

```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

→ To export the Decision Tree

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

→ .dot to .png

* Making predictions

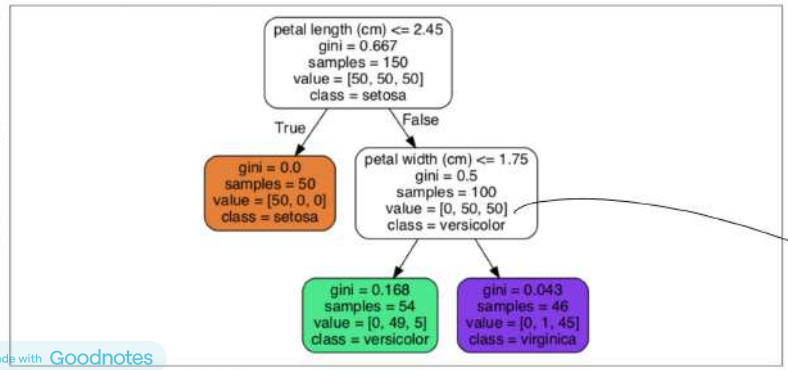


Figure 6-1. Iris Decision Tree

→ Value
no of sample it applies to

* Decision tree require very less data preparation and infact they don't even require feature scaling at all.

gini impurity

$$G_i = 1 - \sum_{k=1}^{N} p_{i,k}^2$$

$p_{i,k}$ = ratio of class k instances among the training instances in i^{th} node.

** Sci-kit learn use CART-Algo, which can only produce binary tree but there are other algo such as ID, which can produce decision tree with nodes that have more than two children.

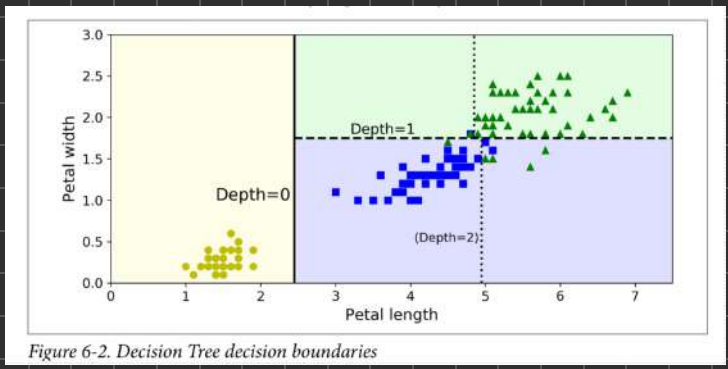


Figure 6-2. Decision Tree decision boundaries

* White box:- intuitive model decision are easy to interrupt like decision Trees.

* Black box:- non-intuitive, decision are not easy to interrupt & and are really complex like neural network and random forest.

CART Training Algo

Classification and regression tree (CART) to train tree.
(Also called "growing trees")

It work by first dividing the training set into 2 subset.
using a single feature k and threshold t_k.

* It chooses (k, t_k) that produces the purest subset.

* CART Cost function.

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} \cdot G_{right}.$$

[$G_{left/right}$ is impurities of the left or right subset.
m is no. of instances left or right.]

* Stopping Condition.

* max - depth
* min - sample - split
* min - sample - leaf
* min - weight - fraction - leaf

CART is greedy algo and give good trees.

* to find, the optimal, the problem is np - complite & require $O(e^x)$ time.

* Computational Complexity

prediction Complexity $\Rightarrow O(\log_2(m))$

train Complexity $= O(n \times m \log_2(m))$

For small data set, less than few thousand. It presort the data, to reduce training time. (presort = true). But it inc training time in large dataset.

Regularization Hyperparameter.

* No. of parameter are not defined before training thus it is non-parametric model ⇒ they can overfit easily.

* linear model which has parameter defined before training are called parametric model.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the Decision Tree: `min_samples_split` (the minimum number of samples a node must have before it can be split), `min_samples_leaf` (the minimum number of samples a leaf node must have), `min_weight_fraction_leaf` (same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances), `max_leaf_nodes` (maximum number of leaf nodes), and `max_features` (maximum number of features that are evaluated for splitting at each node). Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.

* Regression.

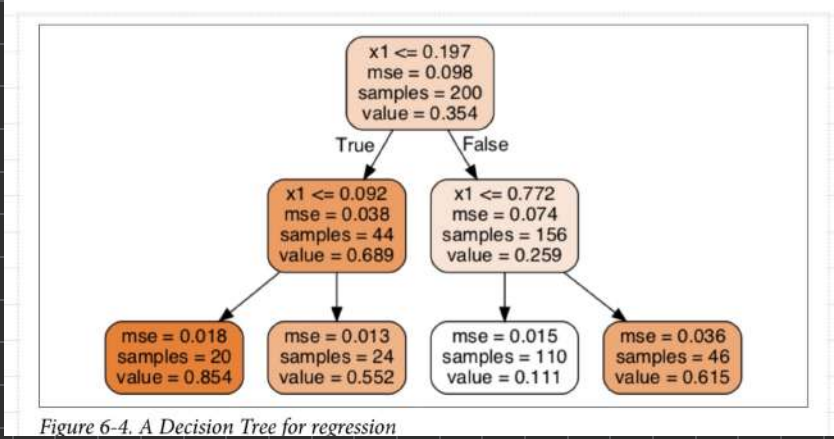


Figure 6-4. A Decision Tree for regression

* instead of class, it predict a value and which is avg of sample in the node and it tries to minimize MSE. error.

* CART cost function for Regression.

$$J(k, t_k) = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right}$$

where

$$\begin{cases} MSE = \sum_{i \in node} (\hat{y}_{node} - y^{(i)})^2 \\ \hat{y}_{node} = \frac{1}{m_{node}} \sum_{i \in node} y^{(i)} \end{cases}$$

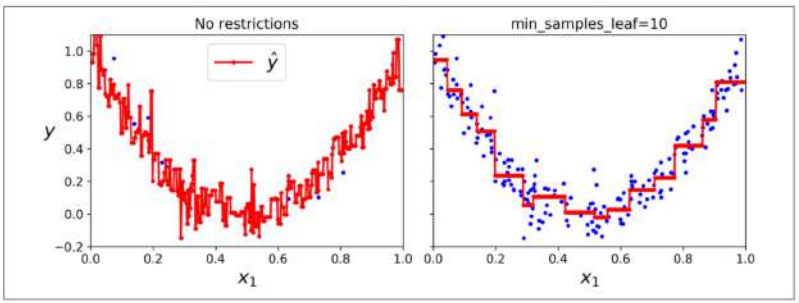
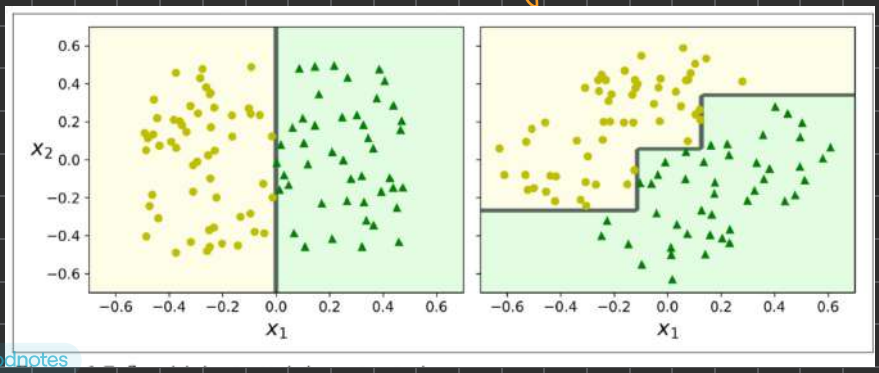


Figure 6-6. Regularizing a Decision Tree regressor

* Regression, DT can easily overfit the data.

* Unstability
  prone to rotation of training set.



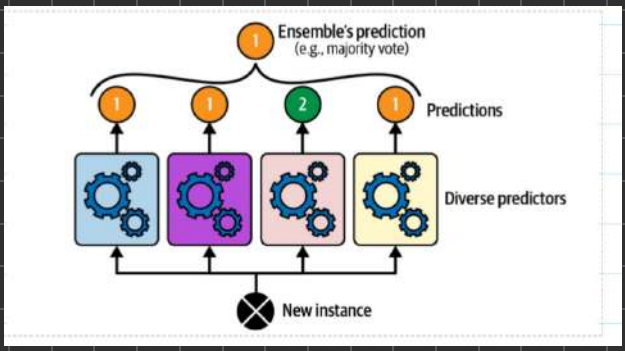
* Sensitive to small variation in training set
(-) The training algorithm used by sklearn is stochastic and you may get very different model for same training data.

(-) Random forest can overcome this overagging prediction over many decision trees prediction

ENSEMBLE LEARNING.

→ Wisdom of Crowd
→ If group of predictor: Ensemble.

- Voting Classifier.
  - Hard voting classifier
    - aggregate the prediction of each classifier and classification with max vote wins.



  - best work when predictor are independent, thus use different algorithm.

- Scikit-learn's Voting Classifier.

```
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

to access individual estimator

For name, clf in voting_clf.named_estimators.items():

2) Soft Voting.

also use probability function from the estimators.

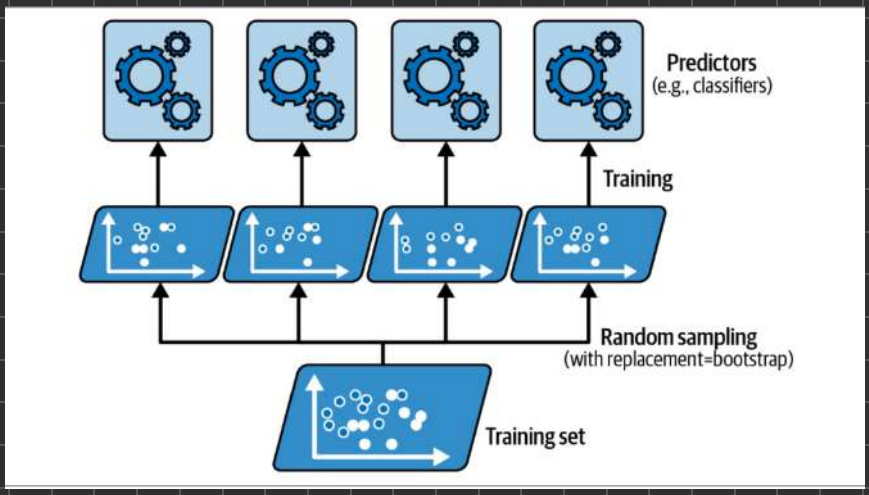
it can be opted as

```
>>> voting_clf.voting = "soft"
>>> voting_clf.named_estimators["svc"].probability = True
>>> voting_clf.fit(X_train, y_train)
>>> voting_clf.score(X_test, y_test)
0.92
```

* Bagging & Pasting.

1) Bagging :- using same training algorithm for every algorithm but using different random subset of training with replacement

2) Pasting :- same without replacement.



*) hard or soft Voting in the model

*) same bais but less variance than a normal models

*) all predictor can be trained on different cpu cores or server.

```python
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

no of CPU Cores

-1 => All

A `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with decision tree classifiers.

For regressor :- Bagging Regressor.

*) Bagging introduce a more diversity in the subset so bagging end up with a slightly higher bais thus predictor being less correlated, so ensemblies variance is reduced thus better model.

to use pasting method in the bagging classifier set bootstrap: false.

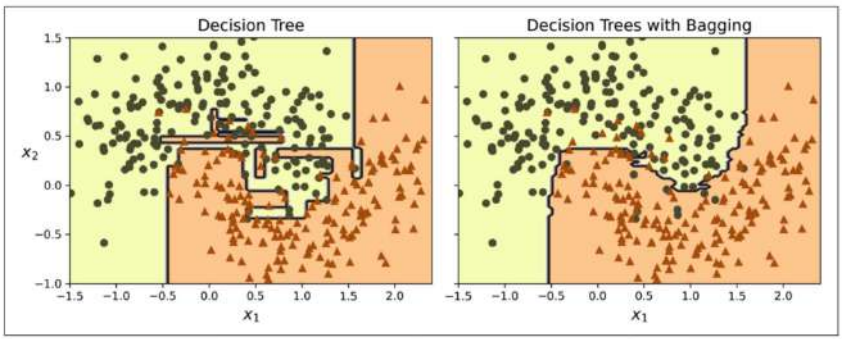


Figure 7-5. A single decision tree (left) versus a bagging ensemble of 500 trees (right)

* Out-of-bag evaluation

in bootstrap sampling

probability of selection of an instance in an sampling one time $\frac{1}{m}$

of not selection = $\left(1-\frac{1}{m}\right)$

and after m-times doing this thing $\left(1-\frac{1}{m}\right)^m$.

$$\lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m = e^{-1} = 0.367$$

selection is ≈ 63%.

So for each predictor around 37% instance are not selected and thus can be used as model evaluation.

* We can also sample no. of feature by hyper parameter max-feature & bootstrap-feature.

* It is useful when you are dealing with high dimension data like images as it can speed up training

* Sampling both instance & feature is called random patches method where as Sampling just feature is called random subspace method.

Random forest.

- ensemble of decision trees
- generally trained with bagging method (with max-sample set to size of training set)

* RFC has all hyperparameter of bagging classifier to control the ensemble & all hyperparameter of a decision true.

* Also, instead of searching for best feature to split, sample in features & choose best from it.

(•) Extra - Trees.

|| Extremely Randomized Trees.

- randomized threshold for each features, in motion the find best possible
- use splitter = "random" in a decision Tree Classifier.

or Scikit-learn's API ⇒ Extra Tree Classifier
or Extra tree Regressor.

* Boosting

- hypothesis boosting

(•) general idea of boosting in to train prediction - sequentially each trying to correct its predecessor.
Most famous boosting method are adaboost & Gradient boosting.

- Adaboost.

new predictor pay a little more attention to the training instances that the predecessors underfit. This results in new predictor focusing more and more on hard class.

It increase the relative weight of mis classified. training instances.

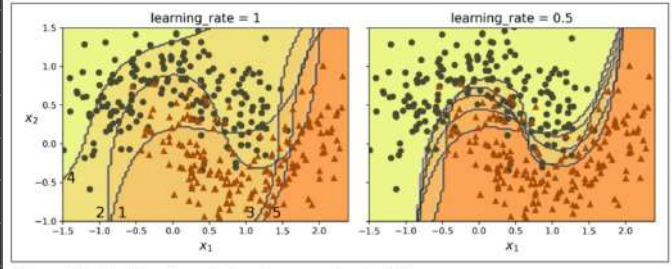


Figure 7-8. Decision boundaries of consecutive predictors

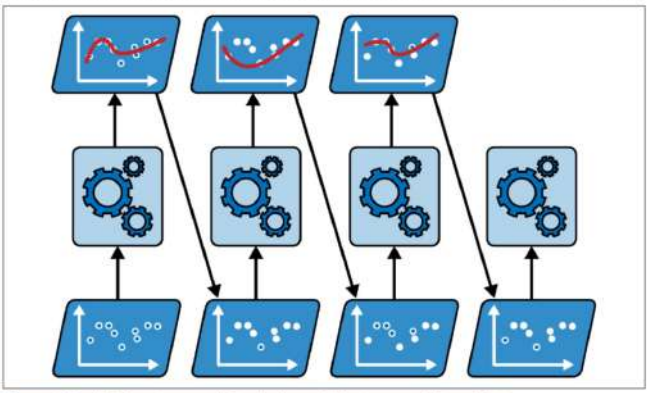


Figure 7-7. AdaBoost sequential training with instance weight updates

* It does not scale well as these can't be parallel training

* All instances weight are more equal to $1/m$.

the weighted error of j^{th} predictor is calculated.

$$r_j = \sum_{\substack{i=1 \\ \hat{y}_i \neq y_i}}^{m} w^{(i)}$$

↳ means wrong prediction

predictor weight.

$$\alpha_j = \eta \log\left(\frac{1 - r_j}{r_j}\right)$$

$$w_i = w_i \exp(\alpha_j)$$

If $\hat{y}_i \neq y_i$ then normalization, such that sum is 1.

Sklearn uses multiclass adaboost.

SAMME.

* If predictor can make class estimator then Sklearn uses SAMME.R

Stage wise additive modelling using a multiclass Exp func.

•) Gradient boosting.

first training the model on training data.
then training it on residual of the data and doing it give time & summing the func.

* for trees, it's called gradient boosted regression trees [GBRT]

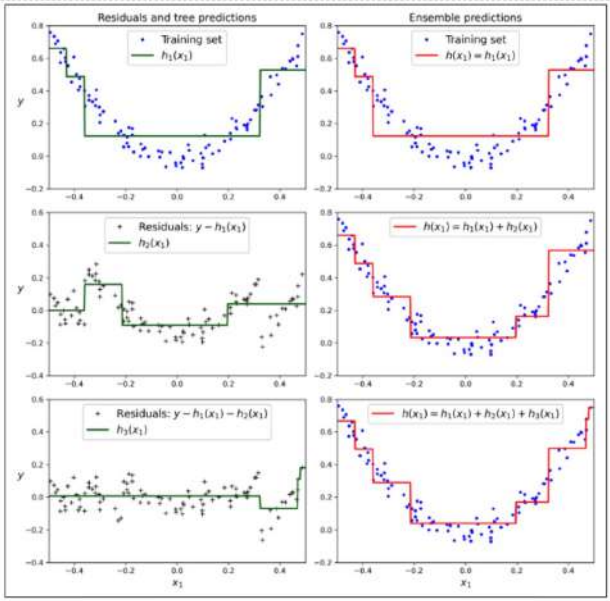


Figure 7-9. In this depiction of gradient boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

Setting small learning-rate you will need more trees to fit. this regularization method called Shrinkage.

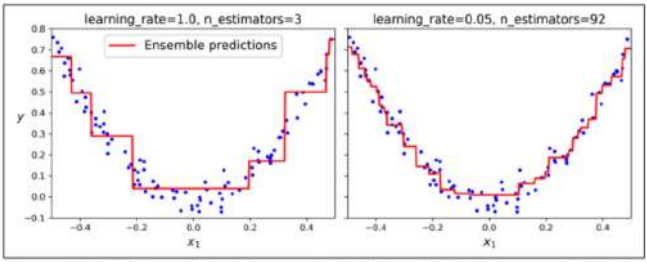


Figure 7-10. GBRT ensembles with not enough predictors (left) and just enough (right)

-> It also has *n_iter_no_change hyper parameters which denote the GBR to stop if test ($n_iter_no_change)

* when it set the model automatically divide training set into training set & validation set.

* It is controlled by validation_fraction hyperparameter which is 10% by default.

* The hyperpara determine maximum performance improvement that is count as negligible (default = 0.01)

* Gradient Boosting Regressor class also has subsample hyperparameters which specify the fraction of training instances to be used to train each tree.

this method is known as stochastic gradient boosting.

-> Stacking

training model then training a Blends -> or (meta_learner) which take in model prediction & gives out a predict.

* We can also take another layer of blender to train & train a find blender.

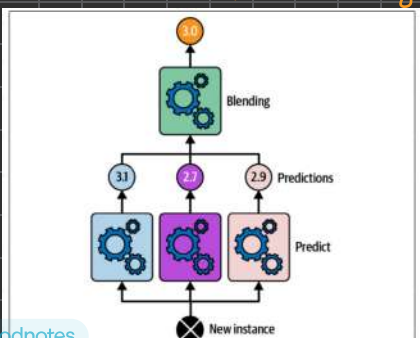
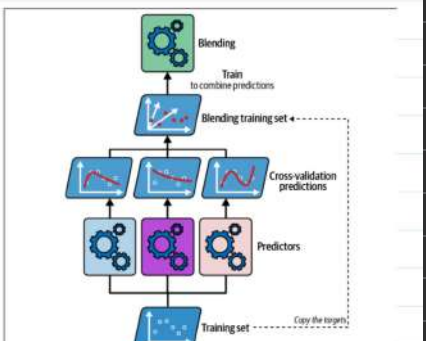


Figure 7-11. Aggregating predictions using a blending predictor



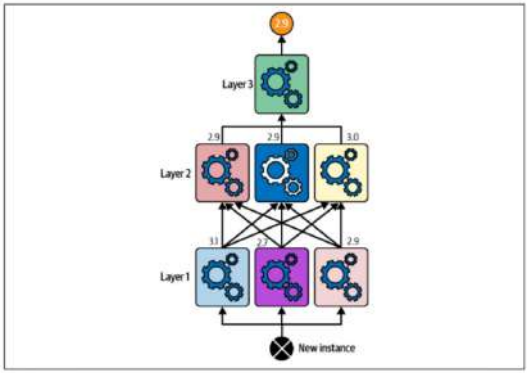


Figure 7-13. Predictions in a multilayer stacking ensemble

(*) If you don't provide final - estimator then Stacking classifier was logistic Regression & " Regressor " Ridge "

DIMENSIONALITY REDUCTION

* To speed up training
* The curse of dimension.
* dimension reduction might improve prediction.

- Projection.

  If all training data lie within or closer a lower dimensional subspaces, then we can take projection.

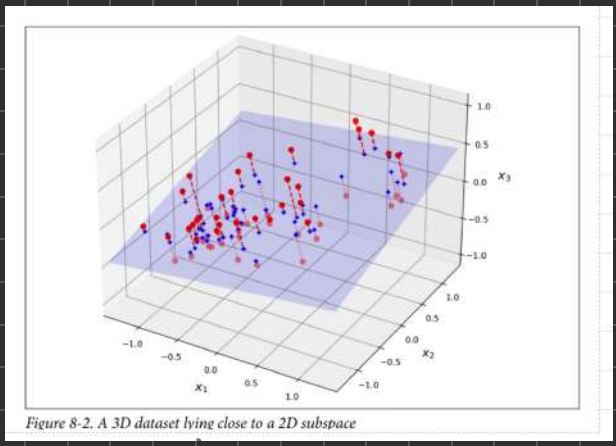


Figure 8-2. A 3D dataset lying close to a 2D subspace

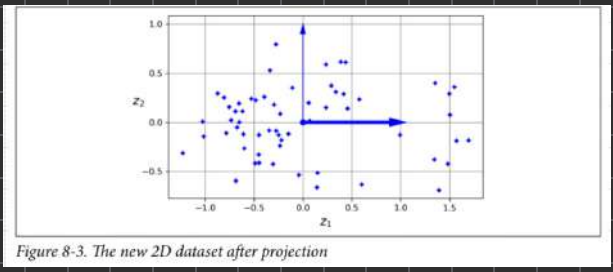


Figure 8-3. The new 2D dataset after projection

* Manifold Learning.

  projection is not always the best approach

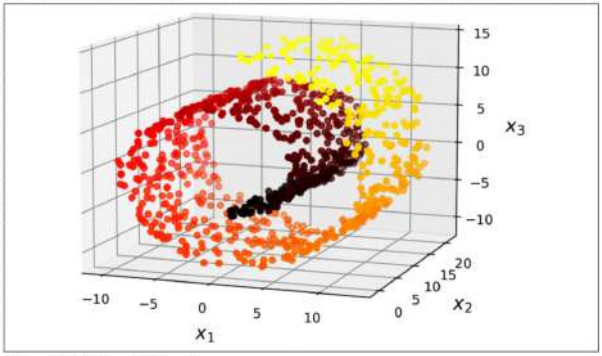


Figure 8-4. Swiss roll dataset

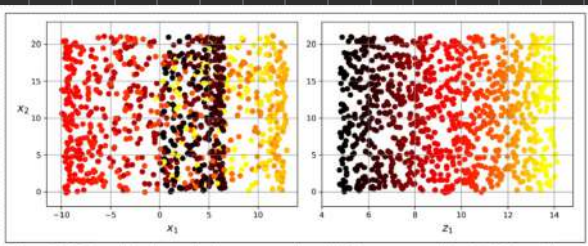


Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

this swiss roll is eg of 2D manifold in 3-D
generally a dimensional manifold in n-d space

is folding of d-d in n-d space, eg swiss roll 2D in 3D

* Many dim - reduction algo work modeling the manifold called as manifold learning. It relies on manifold assumption also called manifold hypothesis.

* Another implicit assumption is also made that the task will be easy in manifold.

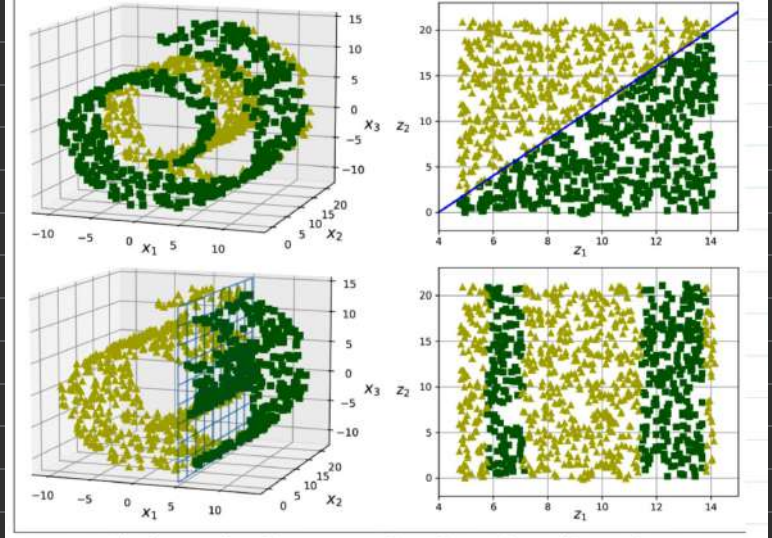


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

training will be easier but model is simple or not depend on the data.

PCA

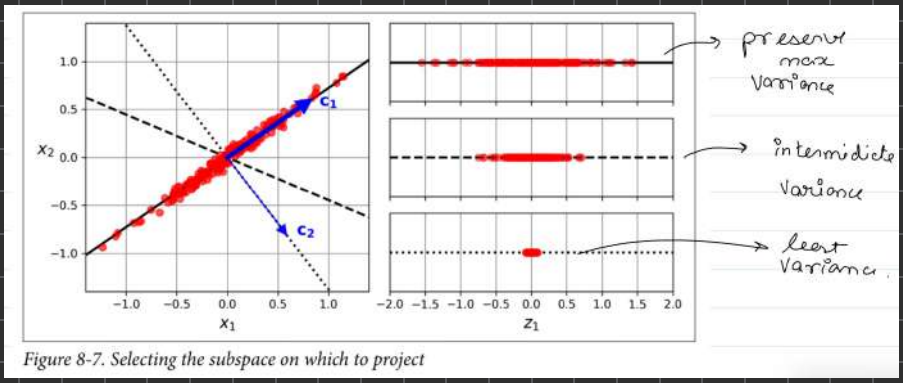


Figure 8-7. Selecting the subspace on which to project

→ to find a hyperplane which preserve max variance

Reason. * to lose less info.

Another justification :- minimize mean squared distance b/w original dataset & its project into data set.

Principal Component.

Axis which account's for largest value of Variance in the training set.

The i^{th} axis is called the i^{th} principal component of data.

⇒ When we do SVD (Singular value decomposition).

We convert training set X into multiplication of $V \Sigma V^T$, where V contain unit vector that define all principal axis.

$$V = \begin{pmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & & | \end{pmatrix}$$

* Projecting Down to d- dimension.

project it to first d principal component

& make sure that projection will preserve as much as variance as possible.

$$X_{d\text{-proj}} = X W_d$$

$$W_d = V[:, d].T$$

* Another method is to plot Explained Variance (cumsum) vs dimension and find its elbow.

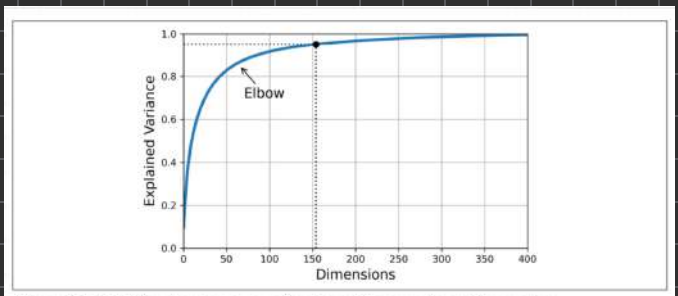


Figure 8-8. Explained variance as a function of the number of dimensions

PCA for Compression.

- while preserving 95% of its variance, dataset is less than 20% of to original size but only 5% of it's variance.

- We can decompress the reduced dataset back to 78, but lost a bit of data (5%).
  MSE of original data & reconstructed data is called reconstruction error.

```
X_recovered = pca.inverse_transform(X_reduced)
```

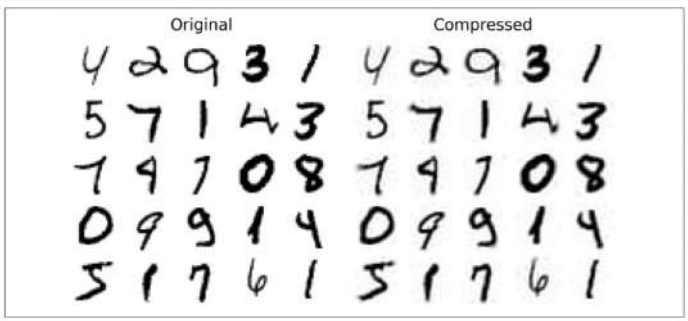


Figure 8-9. MNIST compression that preserves 95% of the variance

$X_{recovered} = X_{d\text{-}proj} W_d^T$

Randomized PCA

It was stochastic Algo called Randomized PCA.
and computational Complexity is $O(m \times d^2) + O(d^3)$.
instead of $O(m \times n^2) + O(n^3)$.
Very time space saving when $d < n$
like 154 out of 784.

By default, `svd_solver` is actually set to `"auto"`: Scikit-Learn automatically uses the randomized PCA algorithm if $\max(m, n) > 500$ and `n_components` is an integer smaller than 80% of $\min(m, n)$, or else it uses the full SVD approach. So the preceding code would use the randomized PCA algorithm even if you removed the `svd_solver="randomized"` argument, since $154 < 0.8 \times 784$. If you want to force Scikit-Learn to use full SVD for a slightly more precise result, you can set the `svd_solver` hyperparameter to `"full"`.

- Incremental PCA.

PCA require whole dataset to fit in memory but IPCA allow you to split the training set into mini batch & feed these in one mini batch. at a time.

Used for large training sets & applying PCA online (on the fly).

For dataset with 10,000 of feature, them training may become much too slow.

- Random projection

$$d \geq \frac{4 \log(m)}{\left(\frac{1}{2}\varepsilon^2 - \frac{1}{3}\varepsilon^3\right)}$$

where ε is the accepted difference b/w MSD of any two instance.

LLE

locally linear embedding (LLE) is a non-linear dimensionality reduction (NLDR) technique.

- works by measuring how each training instance linearly relates to its nearest neighbors. then look for low dimensional representation of the training set where these local relations are best preserved.

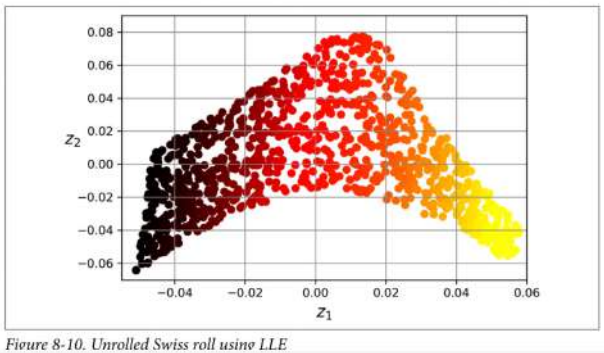


Figure 8-10. Unrolled Swiss roll using LLE

It is not able to maintain distance on a large scale but a great job.

Equation 8-4. LLE step 1: linearly modeling local relationships

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} \mathbf{x}^{(j)} \right)^2$$

$$\text{subject to} \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ n.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 \text{ for } i = 1, 2, \cdots, m \end{cases}$$

Equation 8-5. LLE step 2: reducing dimensionality while preserving relationships

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Wcap is the weight matrix

Z cap is the dimension reduced matrix

Computational complexity:-

$O(m \log(m) n \log(k))$ for finding the k-nearest neighbors, $O(mnk^3)$ for optimizing the weights, and $O(dm^2)$ for constructing the low-dimensional representations. Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

`sklearn.manifold.MDS`
: *Multidimensional scaling* (MDS) reduces dimensionality while trying to preserve the distances between the instances. Random projection does that for high-dimensional data, but it doesn't work well on low-dimensional data.

`sklearn.manifold.Isomap`
: *Isomap* creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances* between the instances. The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes.

`sklearn.manifold.TSNE`
: *t-distributed stochastic neighbor embedding* (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space. For example, in the exercises at the end of this chapter you will use t-SNE to visualize a 2D map of the MNIST images.

`sklearn.discriminant_analysis.LinearDiscriminantAnalysis`
: *Linear discriminant analysis* (LDA) is a linear classification algorithm that, during training, learns the most discriminative axes between the classes. These axes can then be used to define a hyperplane onto which to project the data. The benefit of this approach is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm (unless LDA alone is sufficient).

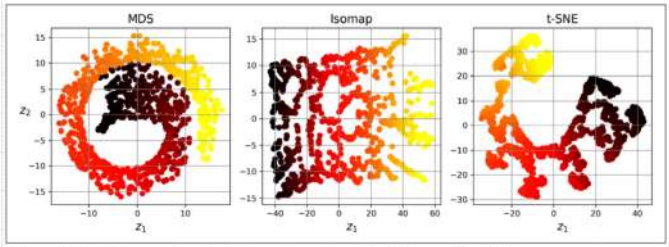


Figure 8-11. Using various techniques to reduce the Swiss roll to 2D

UNSUPERVISED LEARNING

Yann Le cun said "If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be a cherry on cake.

- Unsupervised learning has the potential.

- Clustering.
  group similar instance together into clusters.
  - Best thing for data analysis, customer segmentation, recommended systems, image segmentation, dimensionality reduction

- Anomaly detection. (also called outliner detection)
  - to learn what normal data look like, & detect abnormal instance.
  - In detection of fraud, defective product, identifying new trend or removing outliner from dataset before feeding it to another model.

- Density Estimation.
  to identify the probability density function (PDF) of the random process, that generate the dataset.
  instances in low pdf area are generally outliers.

- Clustering Algorithm.

  Used for :-
  - customer segmentation
  - Data analysis
  - Dimensionality reduction
  - feature engineering
  - Anomaly - detection
  - Semi supervised learning
  - search engine
  - Image segmentation.

K-means (Lloyd - forgy algorithm).

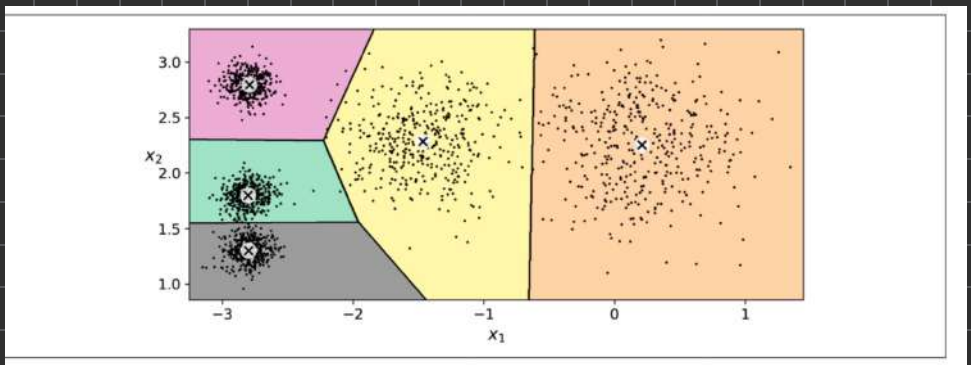
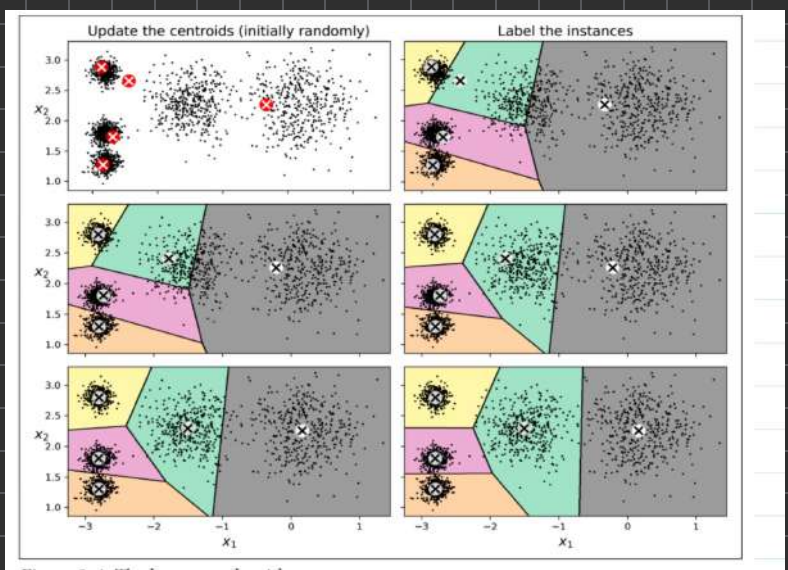


Figure 9-3. k-means decision boundaries (Voronoi tessellation)

- hard clustering :- assigning each point to a class
- Soft clustering :- finding distance from the centroid (scores).

- The algo :- how does it work.

1) random assign cluster.
2) clusterise
3) calculate centres
4) clusterise



- Its algo is guaranteed to converge. It may not converge to optimal solution. (eg. may converge to a local optimum) bcs of unlucky inhaling

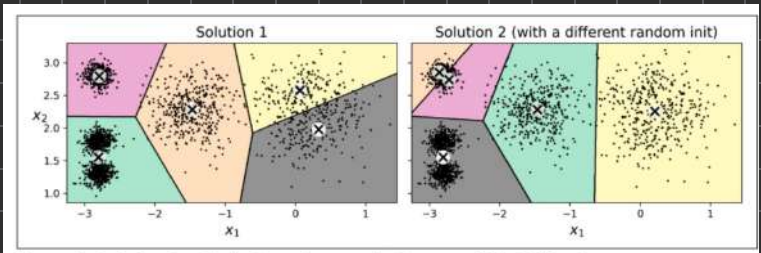


Figure 9-5. Suboptimal solutions due to unlucky centroid initializations

- To improve K-means.
  - → If you know, where the cluster lies you can use this.

→ Train the algo multiple time and keep the best one. no. of random init can be controlled by n-init hyper parameter

to compare solution, it used a performance metric called a model "inertia".

- K mean ++
  - Take a centroid $c^{(i)}$
  - Take another centroid $c^{(i)}$, choosing an instance $x^{(i)}$ with prop $\frac{D(x^{(i)})^2}{\sum_{j=1}^{m} D(x^{(i)})^2}$
  - Sklearn k means use this by default.
- Accelerated K-mean & mini-batch K-mean.
  - Also you can try algorithm = "elkan" It uses triangular inequality to calculate distance sometime. It can be slow or fast. (depend on the dataset).

- Instead of using whole dataset, we can use mini-batch, moving the centroid just slightly at each iteration. (This sped up algo by 3-4 time).

* Used when whole dataset can't fit to the memory.

→ If dataset does'nt fit in memory we can use np.memmap.
→ OR use partial_fit() but you might head to person multiple init & select the best one yourself.

Mini-batch K-mean is faster but its inertia is generally worse.

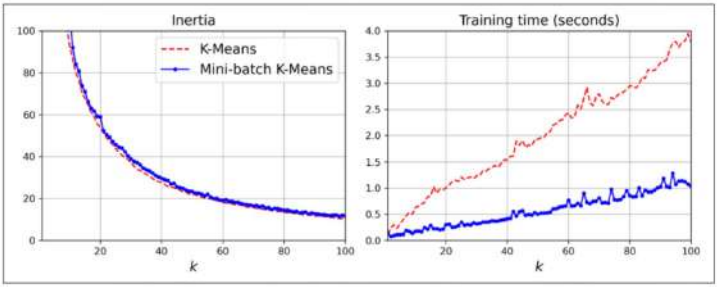


Figure 9-6. Mini-batch k-means has a higher inertia than k-means (left) but it is much faster (right), especially as k increases

(iii) Finding the optimal number of cluster.
- (can't use inertia matrix. as inc K will always inc inertia)

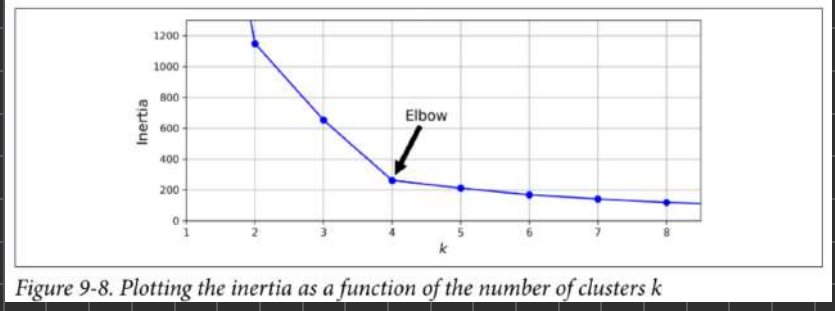


Figure 9-8. Plotting the inertia as a function of the number of clusters k

→ Silhouette score.

which is mean of silhoutte co-efficient over all the instance silhouette co-efficient is equal to $\frac{b-a}{\max(b,a)}$

where a is the mean a is the mean distance to the other instances in the same cluster.
(mean intra-cluster distance)

b is the mean nearest-cluster distance
the mean distance to the instance of next closest cluster
(which minimize b)

* A co-eff close to 1 ⇒ instance are well inside it own cluster & far from other cluster.

* A co-eff close to 1 ⇒ it is close to cluster boundary instance might have been assigned to wrong cluster.

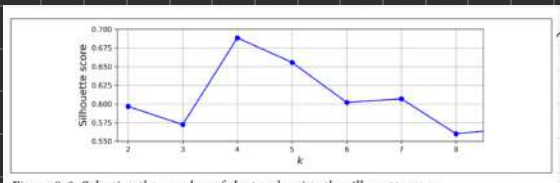
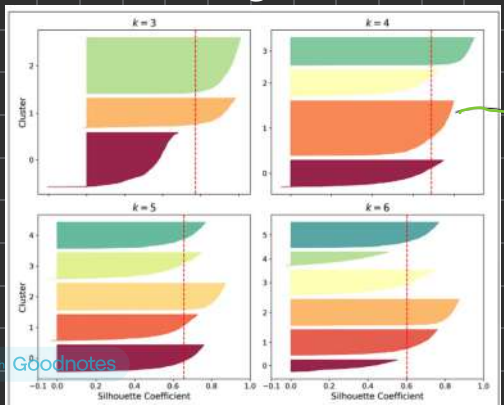


Figure 9-9. Selecting the number of clusters k using the silhouette score

→ 4 is best but 5 is not bad as others. 4/5 can be used

* Another visualization is silhouette diagram (each diagram contain 1 knife per cluster)



→ height represent the no. of instance in the cluster.

width re present sorted silhouette coeff (wider the better)

the dash line is silhouette score

Limit of K-means.

→ multiple inits

→ don't behave well when cluster has different shape, size, density or non spherical shape.

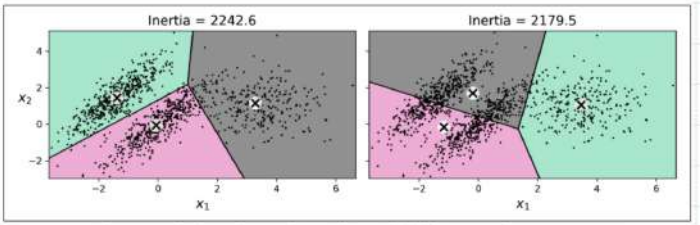


Figure 9-11. k-means fails to cluster these ellipsoidal blobs properly

Using Clustering for Image Segmentation.

- In color segmentation → pixel will similar color get assigned to the same segment. (Eg: checking forest area from a satellite img).
- In semantic segmentation :- all pixel that are part of the same object type get assigned to same segment.
- In instance segmentation:- all pixel that are part of the same individual object are assigned to the same segment.

* Using Clustering for semi-supervised learning.

→ having plenty of unlabelled data but few labelled one

* Achive learning.

To continue improving your model, you can do few round for achive learning.

(i) model is trained on labelled instances so far.
(ii) The instance the model in most uncertain is given to expert for labelling.
(iii) Iterate the process untill the performance improvement stop.

* DB SCAN.

density - based spatial clustering of application with noise (DBSCAN) Algo define Cluster as continous region of high density

- Count how many instances are located within a small distance ε from it. It called ε- neighbourhood.
- If an instance has atleast min - sample instance in its ε- neighborhood., then it is considered a core instance.
- All instance in the neighborhood of a core instance belong to the same cluster (A long sequence of neighborhood core instances form a single cluster).
- Any instance that is not a core instance and doesn't have one in its neighborhood is considered an anomaly.

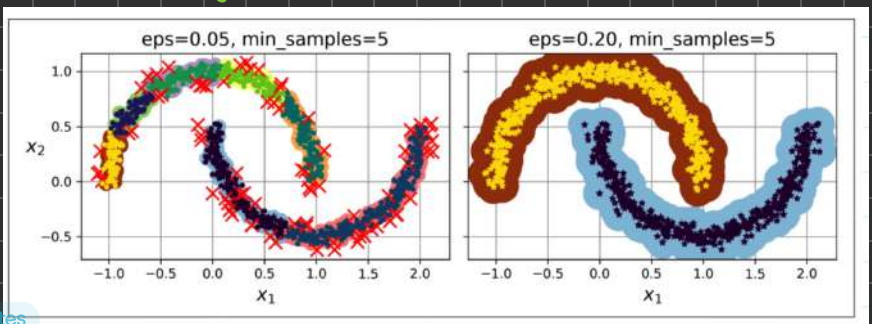


Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```

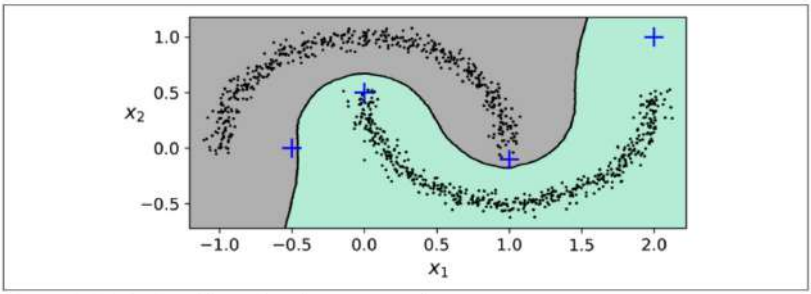


Figure 9-15. Decision boundary between two clusters

HDBSCAN.

HDBSCAN - Hierarchical Density-Based Spatial Clustering of Applications with Noise. Performs DBSCAN over varying epsilon values and integrates the result to find a clustering that gives the best stability over epsilon. This allows HDBSCAN to find clusters of varying densities (unlike DBSCAN), and be more robust to parameter selection.

In practice this means that HDBSCAN returns a good clustering straight away with little or no parameter tuning -- and the primary parameter, minimum cluster size, is intuitive and easy to select.

•> Other Clustering Algo.

Agglomerative clustering

A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If you drew a tree with a branch for every pair of clusters that merged, you would get a binary tree of clusters, where the leaves are the individual instances. This approach can capture clusters of various shapes; it also produces a flexible and informative cluster tree instead of forcing you to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if you provide a connectivity matrix, which is a sparse $m \times m$ matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.

BIRCH

The balanced iterative reducing and clustering using hierarchies (BIRCH) algorithm was designed specifically for very large datasets, and it can be faster than batch k-means, with similar results, as long as the number of features is not too large (<20). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this approach allows it to use limited memory while handling huge datasets.

Spectral clustering

This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces the matrix's dimensionality), then it uses another clustering algorithm in this low-dimensional space (Scikit-Learn's implementation uses k-means). Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

Mean-shift

This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one—the radius of the circles, called the *bandwidth*), and it relies on local density estimation. But unlike DBSCAN, mean-shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is $O(m^2n)$, so it is not suited for large datasets.

Affinity propagation

In this algorithm, instances repeatedly exchange messages between one another until every instance has elected another instance (or itself) to represent it. These elected instances are called *exemplars*. Each exemplar and all the instances that elected it form one cluster. In real-life politics, you typically want to vote for a candidate whose opinions are similar to yours, but you also want them to win the election, so you might choose a candidate you don't fully agree with, but who is more popular. You typically evaluate popularity through polls. Affinity propagation works in a similar way, and it tends to choose exemplars located near the center of clusters, similar to k-means. But unlike with k-means, you don't have to pick a number of clusters ahead of time: it is determined during training. Moreover, affinity propagation can deal nicely with clusters of different sizes. Sadly, this algorithm has a computational complexity of $O(m^2)$, so it is not suited for large datasets.

→ Gaussian Mixture

gaussian mixture model.

- It is probabilistic model that assume that the assume that the instance are generated from a mix of gaussian distribution.
- A instance generated from a single distribution forms a cluster that typically look like an ellipsoid.
- Each cluster can have a different ellipsoid shape, size, density & 0 notation.
- In simplest GMM Variant in gaussian mixture class, you must know the no. K of gaussian distribution.

AN dataset X is assumed to be generated through.

(i) a cluster is randomly picked from K.
the prob of choosing the j^{th} cluster is cluster weight $\phi^{(i)}$
The index of Clusrer chosen for i^{th} instance is noted in z.

(ii) if $z^{(i)} = j$, then location $x^{(i)}$ of the instance is sampled randomly from, gaussian distribution with mean $\mu^{(j)}$ & convariance matrix $\Sigma^{(j)}$

$$x^{(i)} \sim N(\mu^{(j)}, \Sigma^{(j)})$$

Expectation-Maximization (EM) Algorithm

The EM algorithm is used to estimate the parameters of the Gaussian distributions in GMMs. It works similarly to the k-means algorithm with the following steps:

1. **Initialization**: The algorithm initializes the cluster parameters (mean μ, covariance Σ, and weights ϕ) randomly.

2. **Iteration**: The algorithm then repeats two main steps until convergence:

   - **Expectation Step (E-step)**: Assign instances to clusters based on the current parameters. This step estimates the probability that each instance belongs to each cluster.
   - **Maximization Step (M-step)**: Update the cluster parameters based on the assignments from the E-step. This step recalculates the parameters to maximize the likelihood of the observed data given the current assignments.

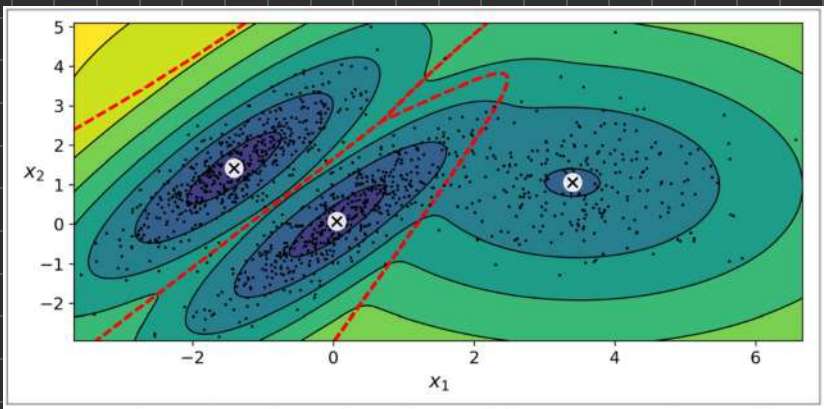


Figure 9-16. Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model

- with many dimension it is hard for EM to learn we can improve it by setting covariance - type to one the following & fixing some parameter.

- "spherical" - all spherical cluster.

- "diag" - can take any size ellipsoid but axis must be parallel to co-ordinate axis.

- tied → all cluster must have same shape size & orientation (all cluster have same covariance matrix)

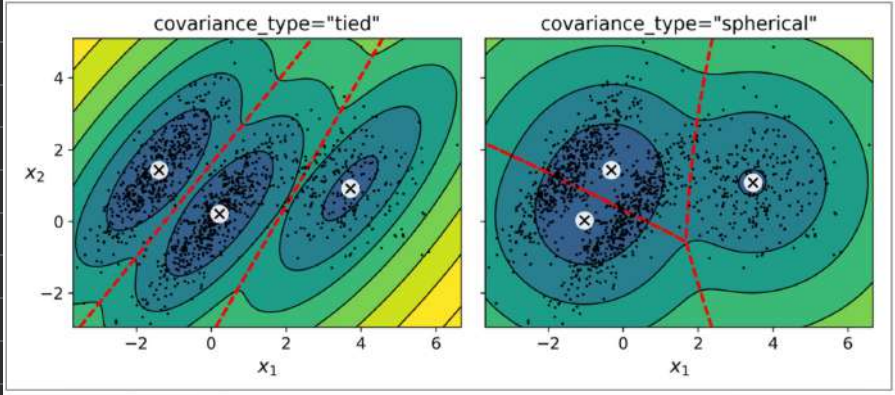


Figure 9-17. Gaussian mixtures for tied clusters (left) and spherical clusters (right)

*) gaussian mixture for anomaly detection.

- instance in low density area can be considered as anomaly we have to set a threshold.

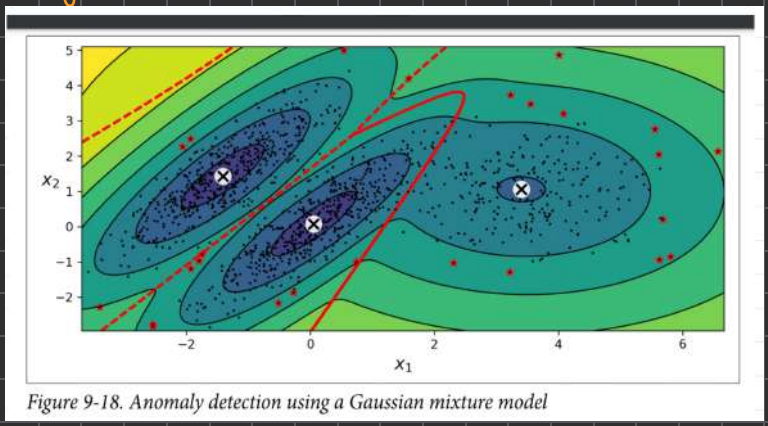


Figure 9-18. Anomaly detection using a Gaussian mixture model

* selecting the number of cluster.
can use BIC or AIC

Bayesian Information Criterion. (BIC) → $\log(m)\,p - 2\log(\hat{L})$
Akaike " " (AIC) → $2p - 2\log(\hat{L})$.

m → no. of instance
p → no. of parameter learned by model
$\hat{L}$ → max value of likelyhood function

Both BIC & AIC penalize model with more parameter & reward the model that fit well.

But if they suggest different model

Likelyhood function.

→ Probability vs likelyhood

→ Probability is how likely a future outcome is, given the model
→ likelyhood → how plausible a set of parameter is given, observed data

PDF → $P(x|\theta)$ other ($\theta = \mu, \sigma$) σ → deviation, μ → centre

Likelyhood func $P(\theta|x)$

all possible values of θ the result can be any positive value.

Given a dataset **X**, a common task is to try to estimate the most likely values for the model parameters. To do this, you must find the values that maximize the likelihood function, given **X**. In this example, if you have observed a single instance x=2.5, the *maximum likelihood estimate* (MLE) of θ is $\hat{\theta}$=1.5. If a prior probability distribution g over θ exists, it is possible to take it into account by maximizing $\mathcal{L}(\theta|x)g(\theta)$ rather than just maximizing $\mathcal{L}(\theta|x)$. This is called *maximum a-posteriori* (MAP) estimation. Since MAP constrains the parameter values, you can think of it as a regularized version of MLE.

Intro To Artificial Neural Network (with Keras).

Biological neurons.

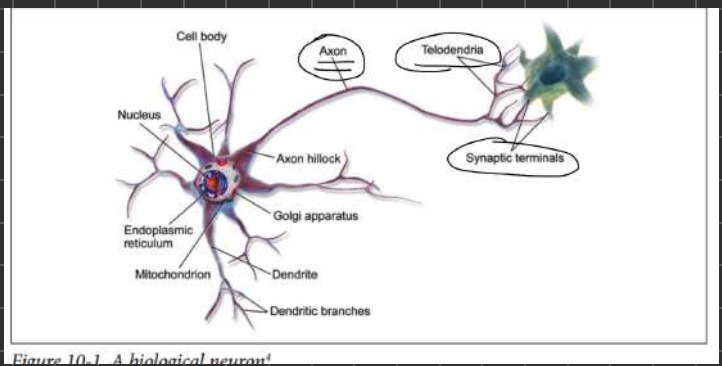


Figure 10-1. A biological neuron[4]

These neuron produce short electrical impulse called action potential (AP) which travel along axon & make synaptic terminal release neuron - transmitive.

When another neuron release a sufficient amount of specific neurotransmitter, if fires it own electrical impulse

Each neuron seem to behave in a single way, but they're organized in a vast network of billion, with each neuron connected to the thousand of others.

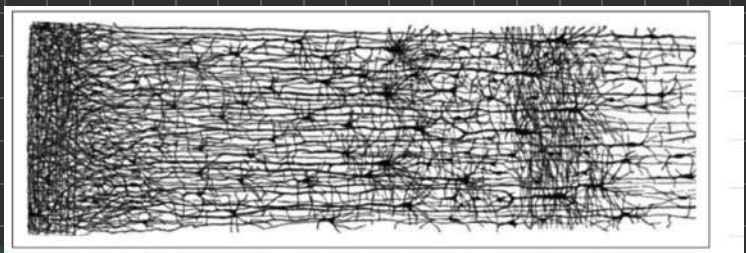


Figure 10-2. Multiple layers in a biological neural network (human cortex)[6]

* Logical Computation with neuron.

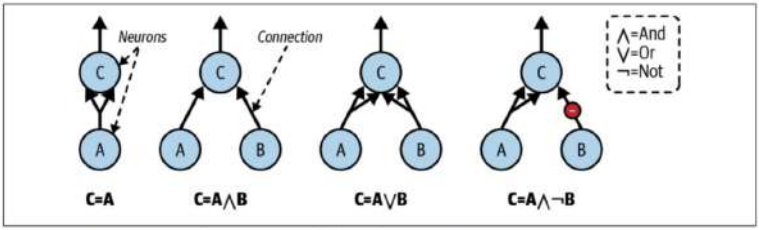


Figure 10-3. ANNs performing simple logical computations

these type of network can perform very well in logical reasoning

Perceptron

Using a slightly different artificial neuron called threshold logical unit (TLU) or linear threshold unit (LTU).

TLU Computer a linear function of its inputs.

$z = w_1 x_1 + w_2 x_2 + w_3 x_3 \cdots w_n x_n + b = W^T x + b$

then apply step-function to the function.

$h_w(x) = \text{step}(z)$.

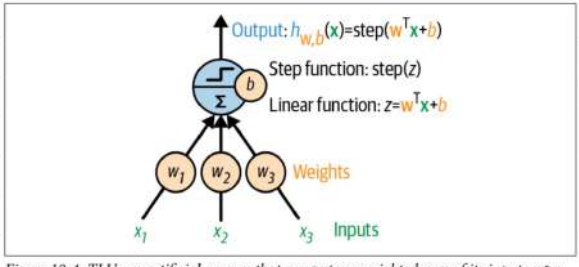


Figure 10-4. TLU: an artificial neuron that computes a weighted sum of its inputs $w^T x$, plus a bias term b, then applies a step function

Commonly used step function are:-

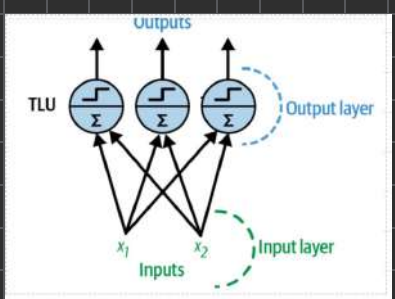
$$\text{heavy side}(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$$

A single TLU can perform simple binary classification.

A perceptron is composed of one or more TLUs organized in a single layer, whose every TLU is connected to every input called Dense layer.

The inputs constitute the input layer.

The layer of TLU is called output layer.



* $h_{w,b}(x) = \phi(x W + b)$
  - activation function
  - weight matrix (one row per input & one column per neuron).
  - bais term (one per neuron).
  - matrix of input features (one row per instances & one column per feature)

* biological neuron train when one interact with other " cell that fire together, wire together.

Perceptron train by similar rule as perceptron learning rule. (weight update)

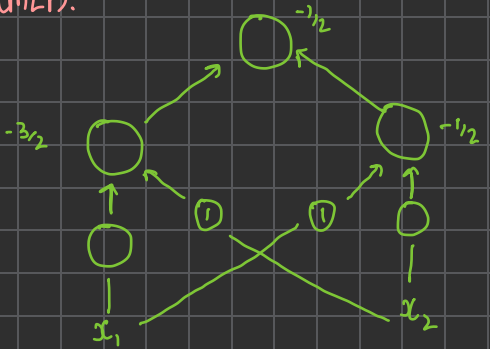
$$w_{i,j}(\text{next step}) = w_{(i,j)} + \eta (y_j - \hat{y}_j)\, x_i$$

- $w_{i,j}$: weight btw i^{th} input & j^{th} neuron
- x_j is the i^{th} input value of current instance.
- y_j: output of j^{th} output for j^{th} output neuron.
- $\hat{y}_j$: target output.

* the decision boundary is linear, so perceptron are incapable of learning complex rules but will converge if there is a linear separable data. It's called perceptron theoram.

Can also use SGD classifier with loss = "perceptron",
learning_rate = "constant"
penalty = None (no regularization)

We were able to remove this problem by using multilayer perceptron (MLP).



* The Multilayer perceptron and Back propagation.

* MLP is composed one input layer, one or more layer of TLU's called hidden layer, and final layer of TLU's called the output layers.

* Layers close to input layer are usually called the lower layer & the one close to the output are usually called upper layers.

[the signal flow only in one direction so this architecture is example of feed forward neural network (FNN)]

When an ANN contain a deep stack of hidden layers is called deep neural network (DNN).

* Back prop.

• Researchers were not able to train MLP. in 1960. they thought of using gradient descent but were not able to calculate gradient of model error with regard to the model parameters.

• In 1970, seppo linnainma created an algo called reverse - mode automatic differentiation (reverse - mood auto - diff).

In just two passes (one forward & one backward), it was able to calculate gradient of the neural network's error with regard to every single model parameter.

* How it works

* It take 1 mini batch at a time & goes through the whole training set multiple time. Each pass is called one Epoch.

* It passed through each layer & layer's data is saved for another pass.

* Error of the model is calculated

* Calculate how much error come from each TLU in output layer

* Il by for each neuron, through back propagation it is calculated How much come from back layer.

* An gradient decent step is followed to tweak, all connection weight.

It is important to initialize all the hidden layers' connection weights randomly, or else training will fail. For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical. In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you *break the symmetry* and allow backpropagation to train a diverse team of neurons.

But for backprop to work properly, the activation function should be changed as step function create a plane graph and its not possible to calculate gradient to word minima in plane graph.

Sigmoid function ⇒ $\sigma(z) = \frac{1}{1+e^{-z}}$

Hyperbolic tangent function ⇒ $\tanh(z) = 2\sigma(2z) - 1$.

Rectified linear unit function ⇒ $ReLU(z) = \max(0, z)$.

Why do we need activation functions in the first place? Well, if you chain several linear transformations, all you get is a linear transformation. For example, if $f(x) = 2x + 3$ and $g(x) = 5x - 1$, then chaining these two linear functions gives you another linear function: $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$. So if you don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

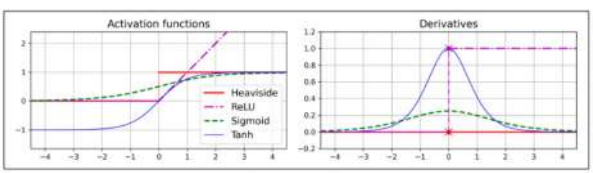


Figure 10-8. Activation functions (left) and their derivatives (right)

* Regression MLP's
  ↳ have output neuron for regressing or multiple for multiple output like co-ordinate regressor

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)  # about 0.505
```

It doesn't use any activation function.

But if you want that output is always +ve you can use ReLU or Soft plus activation function

Soft plus (z) = log (1 + exp (z)).

If you want to have to fall in a certain range.
you can use
Sigmoid for 0 to 1
& tanh for -1 to 1

- MLP Regressor use mean squared error.
  - but you may prefer mean absolute error if there are a lot of outliner.
  - only support MSE.
  - Or use Huber loss which is combination of both.
    l_2 error when error is small then threshold.
    & l_1 when it is greater.

* Classification MLP.

- For binary classifier:- need only 1 single output neuron & sigmoid output function to give output in 0 to 1. can be taken as probability of a class.
- For multi class classifier one output neuron per class.
- For loss function, we use cross entropy loss (x-entropy or log loss for short).

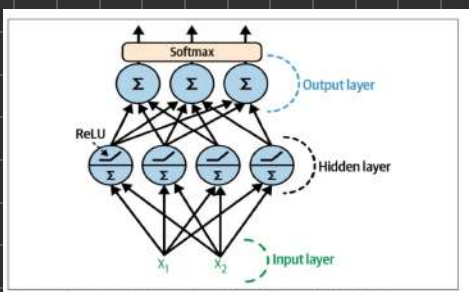


Figure 10-9. A modern MLP (including ReLU and softmax) for classification

* Sklearn has MLP Classifier class in sklearn neural-network package.

Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

* Implementing MLPs with

```
fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]

>>> X_train.shape
(55000, 28, 28)
>>> X_train.dtype
dtype('uint8')

X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test /

tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

All neurons are connected to each neuron in prev layer.

1 layer for input, automatically determine shape as we have size of input.

Convert 2D data to 1D, such that [32, 28, 28] → [32, 784]

⇒ for random set you can also do tf.keras.utils.set_random_seed(), which set random seed for tensor flow & numpy both.

output layer with 10 neuron (1 per class.

you can also do

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

→ instead by adding every single layer.

can also drop input layer. by specify input-shape in flatten.

```
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0

dense (Dense)                (None, 300)               235500

dense_1 (Dense)              (None, 100)               30100

dense_2 (Dense)              (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

you can change this name using name argument in construction.

14 You can also use tf.keras.utils.plot_model() to generate an image of your model.

```
>>> model.layers
[<keras.layers.core.flatten.Flatten at 0x7fa1dea02250>,
 <keras.layers.core.dense.Dense at 0x7fa1c8f42520>,
 <keras.layers.core.dense.Dense at 0x7fa188be7ac0>,
 <keras.layers.core.dense.Dense at 0x7fa188be7fa0>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ...,  0.03859074, -0.06889391],
       [ 0.00476504, -0.03105379, -0.0586676 , ..., -0.02763776, -0.04165364],
       ...,
       [ 0.07061854, -0.06960931,  0.07038955, ..., 0.00034875,  0.02878492],
       [-0.06022581,  0.01577859, -0.02585464, ..., 0.00272203, -0.06793761]],
       dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., ...,  0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

Compile the model.

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

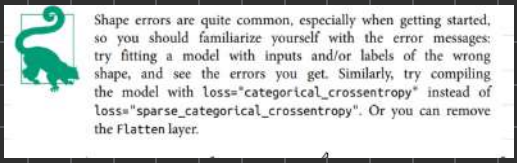
- Sparse - categorical - crossentropy " bcz we have sparse label as each instance has one target probability.
- eg we have one target probability for each class for each instance (eg. one hot head) use " categorical - crossentropy ".
- If we do binary classification or multilabel binary, then we use the " Sigmoid " activation function. instead of the "Soft - max" activation function and we would use the " binary - crossentropy " loss.

If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, use the `tf.keras.utils.to_categorical()` function. To go the other way round, use the `np.argmax()` function with `axis=1`.

When using the `SGD` optimizer, it is important to tune the learning rate. So, you will generally want to use `optimizer=tf.keras.optimizers.SGD(learning_rate=__???__)` to set the learning rate, rather than `optimizer="sgd"`, which defaults to a learning rate of 0.01.

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Epoch 1/30
1719/1719 [==============================] - 2s 989us/step
  - loss: 0.7220 - sparse_categorical_accuracy: 0.7649
  - val_loss: 0.4959 - val_sparse_categorical_accuracy: 0.8332
Epoch 2/30
1719/1719 [==============================] - 2s 964us/step
  - loss: 0.4825 - sparse_categorical_accuracy: 0.8332
  - val_loss: 0.4567 - val_sparse_categorical_accuracy: 0.8384
[...]
Epoch 30/30
1719/1719 [==============================] - 2s 963us/step
  - loss: 0.2235 - sparse_categorical_accuracy: 0.9200
  - val_loss: 0.3056 - val_sparse_categorical_accuracy: 0.8894
```

for validation or you can use validation_split = 0.1 last 10% data for validation before shuffling.



Shape errors are quite common, especially when getting started, so you should familiarize yourself with the error messages: try fitting a model with inputs and/or labels of the wrong shape, and see the errors you get. Similarly, try compiling the model with `loss="categorical_crossentropy"` instead of `loss="sparse_categorical_crossentropy"`. Or you can remove the `Flatten` layer.

- you can also give weight to classes as we weight to classes as we might classes under-represented & some over representing.
  you can by class_weight argument.
  & can give instance weight by sample_weight.

- you can also give sample_weight in validation set by adding them as third item in the validation data as a third data.

* fit() in this case return a history object. which has
  history.params (training parameter)
  history.epoch (list of epoch)
  history.history (dictionary containing losses & extra metrics).

```
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(
    figsize=(8, 5), xlim=[0, 29], ylim=[0, 1], grid=True, xlabel="Epoch",
    style=["r--", "r--.", "b-", "b-*"])
plt.show()
```

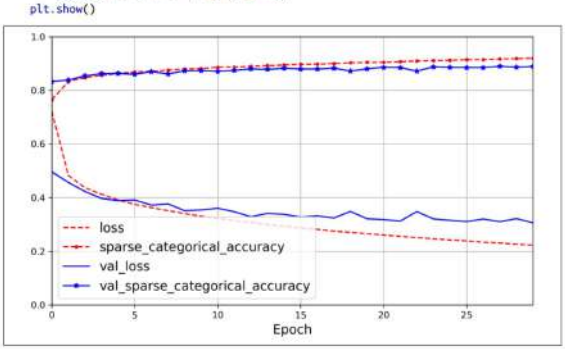


Figure 10-11. Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

```
>>> model.evaluate(X_test, y_test)
313/313 [==============================] - 0s 626us/step
  - loss: 0.3243 - sparse_categorical_accuracy: 0.8864
[0.32431697845458984, 0.8863999843597412]
```

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.02, 0.  , 0.97],
       [0.  , 0.  , 0.99, 0.  , 0.01, 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

If you only care about class with highest estimated probability.

```
>>> import numpy as np
>>> y_pred = y_proba.argmax(axis=-1)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

* Regression MLP using the sequential API.

We use one single neuron as output layer & no activation function loss is MSE & metric is RMSE & Adam optimizer.

and we will be adding a normalization layer instead of flatten.

But its to be called by adapt() before .fit().

```
tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

learns mean & standard deviation

data passes through model.

Building Complex Model Using Functional API (Non-sequential API).

Eg of Non-sequential NN is Wide & Deep Neural Network.

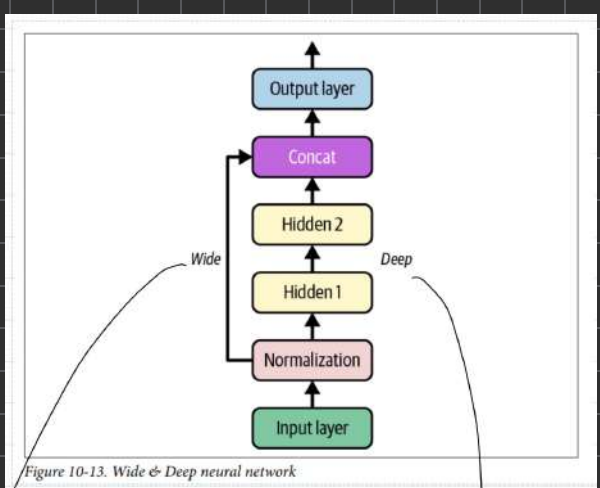


Figure 10-13. Wide & Deep neural network

Wide network helpful to estimate simpler Relation.

Deep Network helpful to estimate Complex relation simpler one might get distorted.

```
normalization_layer = tf.keras.layers.Normalization()
hidden_layer1 = tf.keras.layers.Dense(30, activation="relu")
hidden_layer2 = tf.keras.layers.Dense(30, activation="relu")
concat_layer = tf.keras.layers.Concatenate()
output_layer = tf.keras.layers.Dense(1)
```

→ Defining layers

```
input_ = tf.keras.layers.Input(shape=X_train.shape[1:])
normalized = normalization_layer(input_)
hidden1 = hidden_layer1(normalized)
hidden2 = hidden_layer2(hidden1)
concat = concat_layer([normalized, hidden2])
output = output_layer(concat)

model = tf.keras.Model(inputs=[input_], outputs=[output])
```

→ Arranging layers

→ model pointing input & output

input → for not over shadowing build-in input() func.

All layer work as function, so functional API.

But for case1, where we want to send some data through wide network & some through deep, we can do it by dividing the input like.

```
input_wide = tf.keras.layers.Input(shape=[5])  # features 0 to 4
input_deep = tf.keras.layers.Input(shape=[6])  # features 2 to 7
norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()
norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)
hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)
concat = tf.keras.layers.concatenate([norm_wide, hidden2])
output = tf.keras.layers.Dense(1)(concat)
model = tf.keras.Model(inputs=[input_wide, input_deep], outputs=[output])
```

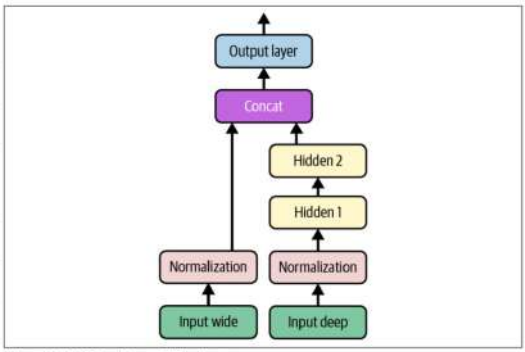


Figure 10-14. Handling multiple inputs

Instead of passing a tuple (`X_train_wide, X_train_deep`), you can pass a dictionary (`{"input_wide": X_train_wide, "input_deep": X_train_deep}`), if you set `name="input_wide"` and `name="input_deep"` when creating the inputs. This is highly recommended when there are many inputs, to clarify the code and avoid getting the order wrong.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])

X_train_wide, X_train_deep = X_train[:, :5], X_train[:, 2:]
X_valid_wide, X_valid_deep = X_valid[:, :5], X_valid[:, 2:]
X_test_wide, X_test_deep = X_test[:, :5], X_test[:, 2:]
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]

norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit((X_train_wide, X_train_deep), y_train, epochs=20,
                    validation_data=((X_valid_wide, X_valid_deep), y_valid))
mse_test = model.evaluate((X_test_wide, X_test_deep), y_test)
y_pred = model.predict((X_new_wide, X_new_deep))
```

- Cases for multiple Output.
  - you want to both locate & classify the main object in a picture
  - Multiple independent task
  - To do regularization, by having a auxiliary output, thus the layer in

Auxiliary output learns something useful.

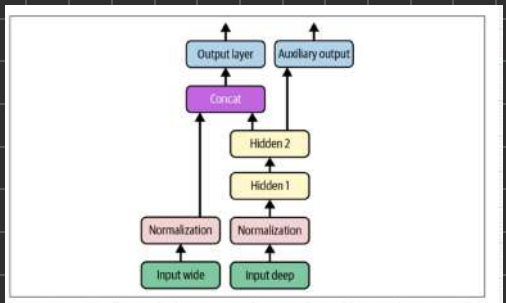


Figure 10-15. Handling multiple outputs, in this example to add an auxiliary output for regularization

```
utput = tf.keras.layers.Dense(1)(concat)
ux_output = tf.keras.layers.Dense(1)(hidden2)
odel = tf.keras.Model(inputs=[input_wide, input_deep],
                      outputs=[output, aux_output])
ptimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
odel.compile(loss=("mse", "mse"), loss_weights=(0.9, 0.1), optimizer=optimizer,
             metrics=["RootMeanSquaredError"])
```

→ Different output

→ loss method & weight.

Instead of passing a tuple `loss=("mse", "mse")`, you can pass a dictionary `loss={"output": "mse", "aux_output": "mse"}`, assuming you created the output layers with `name="output"` and `name="aux_output"`. Just like for the inputs, this clarifies the code and avoids errors when there are several outputs. You can also pass a dictionary for `loss_weights`.

```
norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit(
    (X_train_wide, X_train_deep), (y_train, y_train), epochs=10,
    validation_data=((X_valid_wide, X_valid_deep), (y_valid, y_valid))
)
eval_results = model.evaluate((X_test_wide, X_test_deep), (y_test, y_test))
weighted_sum_of_losses, main_loss, aux_loss, main_rmse, aux_rmse = eval_results

y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

```
y_pred_tuple = model.predict((X_new_wide, X_new_deep))
y_pred = dict(zip(model.output_names, y_pred_tuple))
```

Using the Subclassing API to build dynamic model.

```
class WideAndDeepModel(tf.keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs)  # needed to support naming the model
        self.norm_layer_wide = tf.keras.layers.Normalization()
        self.norm_layer_deep = tf.keras.layers.Normalization()
        self.hidden1 = tf.keras.layers.Dense(units, activation=activation)
        self.hidden2 = tf.keras.layers.Dense(units, activation=activation)
        self.main_output = tf.keras.layers.Dense(1)
        self.aux_output = tf.keras.layers.Dense(1)

    def call(self, inputs):
        input_wide, input_deep = inputs
        norm_wide = self.norm_layer_wide(input_wide)
        norm_deep = self.norm_layer_deep(input_deep)
        hidden1 = self.hidden1(norm_deep)
        hidden2 = self.hidden2(hidden1)
        concat = tf.keras.layers.concatenate([norm_wide, hidden2])
        output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return output, aux_output

model = WideAndDeepModel(30, activation="relu", name="my_cool_model")
```

Saving and Restoring a model.

```
model.save("my_keras_model", save_format="tf")
```

in which ← saves in 'saved model' format in which or saves as directory with several files & subdirectories

saved_model.pb → keras_metadata.pb

↳ extra info needed by Keras.

Variable subdirectory → all the parameter values.

```
model = tf.keras.models.load_model("my_keras_model")
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

Using Callbacks.

```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_checkpoints",
                                                   save_weights_only=True)
history = model.fit([...], callbacks=[checkpoint_cb])
```

to perform before & after each epoch.
eg we have a validation set
Use save - best - only = true.

Only saves whose perform has been best till now.

or can use early stopping callback

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
                                                     restore_best_weights=True)
history = model.fit([...], callbacks=[checkpoint_cb, early_stopping_cb])
```

→ no of epoch in which of improvement doesn't happen it stops.

you can also write custom callback function.

```
class PrintValTrainRatioCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        ratio = logs["val_loss"] / logs["loss"]
        print(f"Epoch={epoch}, val/train={ratio:.2f}")
```

Using Tensor board for visualization.

or can help you visualize the training model & its accuracy, by reading binary log files.

you can make a function to name the files.

```
from pathlib import Path
from time import strftime

def get_run_logdir(root_logdir="my_logs"):
    return Path(root_logdir) / strftime("run_%Y_%m_%d_%H_%M_%S")

run_logdir = get_run_logdir()  # e.g., my_logs/run_2022_08_01_17_25_59
```

it writes training & validation loss & metrics.

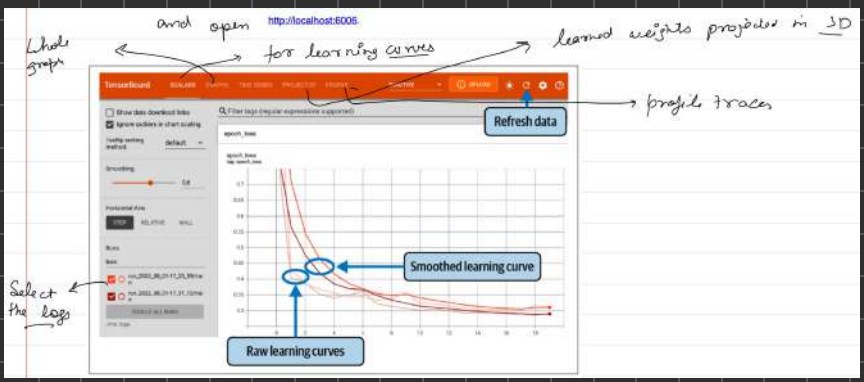
File structure after multiple start.

```
my_logs
├── run_2022_08_01_17_25_59
│   ├── train
│   │   ├── events.out.tfevents.1659331561.my_host_name.42042.0.v2
│   │   ├── events.out.tfevents.1659331562.my_host_name.profile-empty
│   │   └── plugins
│   │       └── profile
│   │           └── 2022_08_01_17_26_02
│   │               ├── my_host_name.input_pipeline.pb
│   │               └── [...]
│   └── validation
│       └── events.out.tfevents.1659331562.my_host_name.42042.1.v2
└── run_2022_08_01_17_31_12
    └── [...]
```

run it directly in Jupyter or Colab using tensor board Extension.

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs
```
Like this

or if everything is running in your machine.



create a custom log file writer low level API in t.f summary.

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(str(test_logdir))
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)

        data = (np.random.randn(100) + 2) * step / 100  # gets larger
        tf.summary.histogram("my_hist", data, buckets=50, step=step)

        images = np.random.rand(2, 32, 32, 3) * step / 1000  # gets brighter
        tf.summary.image("my_images", images, step=step)

        texts = ["The step is " + str(step), "Its square is " + str(step ** 2)]
        tf.summary.text("my_text", texts, step=step)

        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

⇒ Fine tuning neural Network Hyper parameter.

1) by converting keras model to sklearn estimator.
& use Grid search CV or Randomized search CV
you can use keras Regressor & keras Classifier Wrapper
class from SciKeras library

2) OR use Keras Tuner library.
Several tuning strategy, highly customizable & excellent
integration with tensor board.

```python
import keras_tuner as kt

def build_model(hp):
    n_hidden = hp.Int("n_hidden", min_value=0, max_value=8, default=2)
    n_neurons = hp.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2,
                             sampling="log")
    optimizer = hp.Choice("optimizer", values=["sgd", "adam"])
    if optimizer == "sgd":
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    else:
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten())
    for _ in range(n_hidden):
        model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))
    model.add(tf.keras.layers.Dense(10, activation="softmax"))
    model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
                  metrics=["accuracy"])
    return model
```

hyperparameter define

to do random Search

```python
random_search_tuner = kt.RandomSearch(
    build_model, objective="val_accuracy", max_trials=5, overwrite=True,
    directory="my_fashion_mnist", project_name="my_rnd_search", seed=42)
random_search_tuner.search(X_train, y_train, epochs=10,
                           validation_data=(X_valid, y_valid))
```

you can check for best models as

```python
top3_models = random_search_tuner.get_best_models(num_models=3)
best_model = top3_models[0]
```

```
>>> top3_params = random_search_tuner.get_best_hyperparameters(num_t
>>> top3_params[0].values  # best hyperparameter values
{'n_hidden': 5,
 'n_neurons': 70,
 'learning_rate': 0.0004126800823824807,
 'optimizer': 'adam'}
```

API that tell us random_search_tuner's model is gonna be

```
>>> best_trial = random_search_tuner.oracle.get_best_trials(num_trials=1)[0]
>>> best_trial.summary()
Trial summary
Hyperparameters:
n_hidden: 5
n_neurons: 70
learning_rate: 0.0004126800823824807
optimizer: adam
Score: 0.8736000061035156
```

If your are happy with your model train it on full training set for few epoch.

```python
best_model.fit(X_train_full, y_train_full, epochs=10)
test_loss, test_accuracy = best_model.evaluate(X_test, y_test)
```

to fine tune data preprocessing hyper parameter such as model.fit() such as batch_size().

subclass the kt.hyper model.

& define build() & fit().

```python
class MyClassificationHyperModel(kt.HyperModel):
    def build(self, hp):
        return build_model(hp)

    def fit(self, hp, model, X, y, **kwargs):
        if hp.Boolean("normalize"):
            norm_layer = tf.keras.layers.Normalization()
            X = norm_layer(X)
        return model.fit(X, y, **kwargs)
```

```python
hyperband_tuner = kt.Hyperband(
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,
    max_epochs=10, factor=3, hyperband_iterations=2,
    overwrite=True, directory="my_fashion_mnist", project_name="hyperband")
```

Similar as having Random Search CV
only keep 1/factor model after few epochs.

```
root_logdir = Path(hyperband_tuner.project_dir) / "tensorboard"
tensorboard_cb = tf.keras.callbacks.TensorBoard(root_logdir)
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=2)
hyperband_tuner.search(X_train, y_train, epochs=10,
                       validation_data=(X_valid, y_valid),
                       callbacks=[early_stopping_cb, tensorboard_cb])
```

This is also Kt. bayesian optimization.
which uses a probabilistic model called a gaussian process.

have two hyper parameter.
α → noise you need expect in the performance measure.
across all trial (default 10.4)
β → how much you want the algo to explore
default (2.6).

```
bayesian_opt_tuner = kt.BayesianOptimization(
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,
    max_trials=10, alpha=1e-4, beta=2.6,
    overwrite=True, directory="my_fashion_mnist", project_name="bayesian_opt")
bayesian_opt_tuner.search([...])
```

No. of hidden layer.
Hidden layer help recognize pattern.
lower layer recognize low level pattern
& higher layer recognize high level pattern.

this hierarchical architecture help DNN converge faster to a good solution, but improve ability to generalize to new dataset.

Can also help in transfer learning
transferring lower NN layer to do a different but similar task.

No of Neurons per hidden layer

It common practice to make a pyramid

like

for MNIST 784 → 300 → 200 → 100 → 10

But it's found that equal No of Neurons in each layer perform better & there is only 1 hyperparameter to train.

Learning rate

The learning rate is arguably the most important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learning rate above which the training algorithm diverges, as we saw in Chapter 4). One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g., 10^{-5}) and gradually increasing it up to a very large value (e.g., 10). This is done by multiplying the learning rate by a constant factor at each iteration (e.g., by $(10 / 10^{-5})^{1/500}$ to go from 10^{-5} to 10 in 500 iterations). If you plot the loss as a function of the learning rate (using a log scale for the learning rate), you should see it dropping at first. But after a while, the learning rate will be too large, so the loss will shoot back up: the optimal learning rate will be a bit lower than the point at which the loss starts to climb (typically about 10 times lower than the turning point). You can then reinitialize your model and train it normally using this good learning rate. We will look at more learning rate optimization techniques in Chapter 11.

Optimizer

Choosing a better optimizer than plain old mini-batch gradient descent (and tuning its hyperparameters) is also quite important. We will examine several advanced optimizers in Chapter 11.

Batch size

The batch size can have a significant impact on your model's performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently (see Chapter 19), so the training algorithm will see more instances per second. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in GPU RAM. There's a catch, though: in practice, large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. In April 2018, Yann LeCun even tweeted "Friends don't let friends use mini-batches larger than 32", citing a 2018 paper[21] by Dominic Masters and Carlo Luschi which concluded that using small batches (from 2 to 32) was preferable because small batches led to better models in less training time. Other research points in the opposite direction, however. For example, in 2017, papers by Elad Hoffer et al.[22] and Priya Goyal et al.[23] showed that it was possible to use very large batch sizes (up to 8,192) along with various techniques such as warming up the learning rate (i.e., starting training with a small learning rate, then ramping it up, as discussed in Chapter 11) and to obtain very short training times, without any generalization gap. So, one strategy is to try to using a large batch size, with learning rate warmup, and if training is unstable or the final performance is disappointing, then try using a small batch size instead.

Activation function

We discussed how to choose the activation function earlier in this chapter: in general, the ReLU activation function will be a good default for all hidden layers, but for the output layer it really depends on your task.

Number of iterations

In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

Training Deep Neural Network.

* Problem you might face while training DNN with 10s of layer with 100s of neuron

(i) Gradient growing ever smaller or larger while flowing backward through the DNN during training. Both of this problem make lower level very hard to train.

(ii) Might not have enough training data to train the model.

(iii) Training is very slow

(iv) With million of parameter, overfitting of training set might happen

* The Vanishing / Exploding gradient problem
  - As we progress down to lower layer in backprop algo, the error gradient which has been parsed through layer, become smaller & smaller.
  - or the gradient can become larger & larger (in case of recurrent neural network). & the algo diverge.

- Moore law - Number of transistor in an integrated circuit doubles about every two years.