Notes on: "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow"

Alexandre De Spiegeleer

January 7, 2021

1 The machine learning landscape

ML is nothing more than fitting a model to the known data!

1.1 Sturcture of a project

1. study the problem
2. Train the ML algo (with lots of data)
3. Solution
4. Check the solution and better understand the problem

→ try to improve model from the observed possible problems

Examples:

- Detection of signals in images using CNNs
- Classifying articles: NLP

1.2 Types of ML

Different categories of ML algorithms:

Properties	**Descrition**	**Examples**
Supervised	Need laballed training data	kNN, Linear Reg., Logistic Reg., SVM, Decision Tree, Random Forest, NN...
Unsupervised	No labelled data, the algorithm puts similar data together	Clustering: k-means, DBSCAN, HCA. Dimensional reduction: PCA, kernel PCA, LLE, t-SNE. Anomaly detection: One-class SVM, Isolation Forest
Semi-Supervised	Both labelled and not labelled data	Deepd belief network, restricted Boltzman machine
Reinforcment Learning	In the learning process give rewards or penalites depending on the action done	
Batch Learning	Trains using ALL data → If needs to retrain, need to train on all the data again	
Online Learning	Train by mini-batches: Train incrementally and can start again from a previous minibatch run	
Instance Based	On new data, check distance to known data and assign same output	
Model-based	Use the data to make predictions / interpolates to new data	

1.2.1 Unsupervised Learning

Feature Extraction Combines related features into a better one (e.g. using PCA)

Anomaly Detection Find anomalies in dataset(e.g. removing some outliers)

Association rule learning Discover relationships in large datasets

1.2.2 semi-supervised Learning

Often much data but only a few are labelled

→ Often combinaison of supervised and unsupervised algo.

1.2.3 Reinforcement Learning

Trains an agent by giving rewards and penalties to obtained the best policy

1.3 Challenges

1.3.1 data

Lack of data ML requires lots of data (mininmum thousands of examples).
Note that models performances increase with number of data.
→ Choice to work on model or gather more data!

Non representative data Data used for training must include similar data to those for predictions. (Poor extrapolation capacity)

Sampling Bias non representative data because the sampling is flawed.
Bias can appear if the proportion of data in different classes are not representative.

Poor data quality if errors, outliers and noise in data

Irrelevant features There should be enough relevant features and not much crap.

1.3.2 Fitting

Overfitting The model fits too well the training data. Occurs because model too complex compared to data → lose predictability.

Underfitting Model too simple compared to the data to be represented e.g. Linear fit on a non-linear problem

Solutions to overfitting:

- *Regularization*
  → Making the model simpler to avoid by adjusting hyperparameter
- *Hyperparameter*
  → Parameters of the learning algorithm that dictates the amount of regularization.

Solutions to underfitting

- Select a better model
- Have better features
- reduce the constraints on the model (e.g. hyperparameters)

1.4 Evaluating performance

Split the data into training set (80%) and testing set (20%). Evaluation on the test set gives an estimate of the error on unseen data. When training multiple models on the training set and testing them on the test sets, our selection of the "best" model is which model best fits the test data set. → Model cannot necessarily be good on new data. Thus, keep a validation set.
Thus, there are 3 sets of data: Training, Validation and Test

1. Train multiple models with different Hyperparameter on the training set
2. Select the model that performs best on the validation set.
3. Verify that the model is indeed good on the test set
4. Deploy

It is common to split the training set to train several models on sub-sets of the training set and train one final model on the whole training set.
If the validation set is too small → imprecise evaluation of performances. The validation set cannot be too large either (compared to training set).
→ use *cross-validation*: it uses several small validation sets. Each model is evaluated once per validation set. You can then average the results of the model on the different smaller validation sets.

1.5 Data mismatch

If there are two types of data in the training set (e.g. not from the same source). This may cause problem in the predictions. How to know if overfitting or problem with the data?
Hold part of the training set (data from one of the source) and see how the model performs on that. If it performs well → the error comes from the mismatched data If it performs poorly → the error comes from the overfitting
If it is because of the data mismatched, is it possible to preprocess these to be more like the rest of the data?

2 End-to-end machine learning project

Main steps of a machine learning project:

1. Look at the big picture
2. Get the data
3. Discover and visualise the data
4. Prepare the data for machine learning algorithms
5. Select a model and train it
6. Fine-tune the model
7. Present the solution
8. Launch, monitor and maintain the system

2.1 Look at the big picture

Get info on the problem to be solved e.g.

- Objectives?
- input and output of the model? (i.e. features and regression/categories?)
- Overall project's pipeline
- Current status and precision → idea for the aimed accuracy
- Evaluate what the model needs: multiple regression (if multiple features), univariate/-multivariate regression (if one/several quatities to predict), continuous flow of data or not, size of the training set, ...
- Select an error measurement
- Check that the assumptions that have been made are reasonable

2.2 Get the data

For reproducibility, it is best that everything is scripted, from the download of the data to the final product.

Create virtual environment

```
python -m virtualenv .venv
```

Download the data

```
import os
import tarfile
import urllib.request

def fetch_data(url, data_path, data_file):
  os.makedirs(data_path, exist_ok=True)
  tgz_path = os.path.join(data_path, data_file)
```

```
urllib.request.urlretrieve(url, tgz_path)
data_tgz = tarfile.open(tgz_path)
data_tgz.extractall(path=data_path)
data_tgz.close()
```

Load the data

```
import pandas as pd

data = pd.read_csv(data_path)
```

Quick info on the data

```
# pandas functions
data.head()
# Info about attributes and number of entries
data.info()
# Get the attribute_str data
data["attribute_str"]
# Counts occurences of the categories in category_str
data["category_str"].value_counts()
# count, mean, ...
data.describe()
```

Distribution of the features

```
import matplotlib.pyplot as plt

data.hist(bins=50, figsize=(20,15))
plt.show()
```

Now we want to get a test set that we will not look at until we have selected the model and we a ready for release. The test gives an indication for the error the model will have on the new data it has never seen (the actual new data without label).

There are different ways to create the test set. One must be careful in the way the train and test sets are created.

- Indeed, we cannot just use random instances that change everytime we run the code. We must use a way that always uses the same instances, even if new data are added! Note that if no new data are added, the problem is simpler. This can be done by creating a unique idea for each instance and spliting by id.
- If a category is particularly important for the prediction, we need to keep the right propertions of this category in the train set and the test set. This is done using stratified sampling:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
                               random_state=42)
# It will split the data following the proportions of
# the categories in category_str
```

```
# Note that if it is not a category but a continuous value,
# one can create a new features which is a bined version
# of the continuous data
for train_index, test_index in split.split(data,
                                    data["category_str"]):
  strat_train_set = data.loc[train_index]
  strat_test_set = data.loc[test_index]

data = strat_train_set
```

Now the proportions are correct. If a category was created, delete it.

2.3 Visualise the data

It is now time to visualise the data and better understand them.
There are many plots and ways to investigate the data, here a few examples:

Scatter Plot

```
data.plot(kind="scatter", x="cat_1", y="cat_2", alpha=alpha,
      s=cat_3, label=label, c=cat_4, cmap=plt.get_cmap("jet"),
      colorbar=True)
      plt.legend()
```

Correlation Correlation between pairs of attributes

```
corr_matrix = data.corr()
corr_matrix["label_var"].sort_values(ascending=False)
```

Scattered matrix plot Scatter of each attributes

```
from pandas.plotting import scatter_matrix

# Possibly too many attributes -> reduce
attributes = [cat1, cat2, cat3]
scatter_matrix(data[attributes], figsize=(12, 8))
```

Combining attributes Sometimes a combination of attributes is better than the attributes separately.

```
data["new_var"] = data["var_1"] / data["var_2"]
# And check the new correlation and hope it's better
corr_matrix = data.corr()
corr_matrix["label_var"].sort_values(ascending=False)
```

2.4 Preparing the data for machine learning

Create functions to automatise the treatment of the data. Preferably in a way that it is general and can be re-used later on.

Data and labels

Start by separating the data and the labels for the train set.

```
data = strat_train_set.drop("label_to_predict", axis=1)
data_labels = strat_train_set["label_to_predict"].copy()
```

Missing Values

If there are missing values, there are different methods

```
# Gets rid of the entities which lack the value in
# cat_to_drop
data.dropna(subset=["cat_to_drop"])
# Get rid of the whole attribute
data.drop("cat_to_drop", axis=1)
# Replace by median value in the whole dataset
median = data["cat_to_fill"].median()
data["cat_to_fill"].fillna(median, inplace=True)
```

This can also be done using sklearn toolbox

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
# need to remove categorical attributes
data_numerical = data.drop("categorical_att", axis=1)
imputer.fit(data_numerical)
# Look at the medians:
imputer.statistics_
# Create a numpy array of transformed data with
# filled values
X = imputer.transform(data_numerical)
data_treated = pd.DataFrame(X,
        columns=data_numerical.columns,
        index=data.data_numerical.index)
```

Categorical Attributes

- replace them by integers from 0 to number of categories-1 and use that for training.

  ```
  from sklearn.preprocessing import OrdinalEncoder
  ordinal_encoder = OrdinalEncoder()
  data_cat = data[["categorical_attribute"]]
  data_cat_encoded = ordinal_encoder.fit_transform(data_cat)
  ```

  There is a problem! The numbers have a relation between each other (bigger/smaller) and that property is not necessarily there in the categorical attributes.

- Instead create a new *onehot* attribute for each of the original categories in the categorical_attribute.

  ```
  from sklearn.preprocessing import OneHotEncoder
  cat_encoder = OneHotEncoder()
  # This will create the new attributes.
  # There will be as many new attributes as there were
  # categories in categorical_attribute.
  data_cat_1hot = cat_encoder.fit_transform(data_cat)
  ```

Custom Transformers

Create own transformers that have *fit* and *fit_transform*. This is usefull when creating a pipeline. It can be done by create a new class.
If it inherits from TransformerMixin, *fit_transform()* gets created automatically. If it also inherits BaseEstimator, there are two more methods: *get_params()* and *set_params()*

```
from sklearn.base import BaseEstimator, TransformerMixin
class CombinedAttributesAdder(BaseEstimator, TransformMixin):
def fit(self, X, y=None):
  # Fit the data X i.e. get the values out of the data
  # and save what must be saved
  return self
  def transform(self, X, y=None):
  # apply to the data the fit by using the saved values
  return

  # Which can be used:
  attr_adder = CombinedAttributesAdder()
  data_extra_attribs = attr_adder.fit_transform(data.values)
```

Feature Scaling

The range of values between the fields can vary a lot.
The algorithm learns better with same range of values for each attribute There are two typical metods: min-max scaling and standardization

```
from sklearn.preprocessing import MinMaxScaler,
                                  StandardScaler

scaler = MinMaxScaler()   # or StandardScaler()
data_scaled = scaler.fit_transform(data)
```

Transformation pipeline

To simplify the consecutive transformation of the data.
Numerical pipeline:

```
from sklearn.pipeline import Pipeline
num_pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy="median")),
            ('attribs_adder', CombinedAttributesAdder()),
            ('std_scaler', StandardScaler())
            ])
data_tr = num_pipeline.fit_transform(data)
```

If numerical and categorical attributes:

```
from sklearn.compose import ColumnTransformer
num_attribs = list(data_numerical)
cat_attribs = ["cat_attributes"]
full_pipeline = ColumnTransformer([
      ("num", num_pipeline, num_attribs),
      ("cat", OneHotEncoder(), cat_attribs)
```

```
        ])
    data_prepared = full_pipeline.fit_transform(data)
?
```

2.5 Select and Train a model

Training and evaluating on the training set

For example, it is easy to train a model:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(data_prepared, data_labels)
```

It can then be tried on the training data:

```
# Start by transforming the data through the pipeline!
train_data_prepared = full_pipeline.transform(train_data)
print("Predictions:", lin_reg.predict(train_data_prepared))
print("Labels:", list(train_data_labels))
```

Error measure using RMSE:

```
from sklearn.metrics import mean_squared_error
data_predictions = lin_reg.predict(train_data_prepared)
lin_mse = mean_squared_error(train_data_labels)
lin_rmse = np.sqrt(lin_mse)
print(lin_rmse)
```

This can give an idea if the model overfitted or underfitted the data during training. If underfitting, 3 solutions:

- Chose a more powerful model
- Feed the model better features
- Reduce the constraints on the model (remove regularization)

If overfitting, solutions:

- Simplify the model
- Constrain the model (regularize it)
- Get more data

More complex model:

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(train_data_prepared, train_data_labels)
```

Can be evaluated on training set using RMSE. If $rmse = 0 \rightarrow$ overfitting! How can we be sure of overfitting? (**We cannot use the test set!! This is only to be used in the very end!**) We split the train set into train set and validation set.

Better evaluation using cross-validation

A way to test whether the model is performing well would be to split the training set into a smaller training set and a validation set.
The different models could be trained on the smaller training set and evaluated on the evaluation set.
An alternative is to use *K-fold cross-validation*:

- it splits the training set into 10 subsets (folds)
- then it trains the model 10 times, everytime using a different fold for the validation and the other 9 folds as the training data.

```
from sklearn.model_selectoin import cross_val_score

scores = cross_val_score(model, train_data_prepared, train_data_labels,
            scoring="neg_mean_squared_error", cv=10)
model_rmse_scores = np.sqrt(-scores)

def display_scores(scores):
    print("Scores: " scores)
    print("Mean: ", scores.mean())
    print("Standard deviation: ", scores.std())

display_scores(model_rmse_scores)
```

Another model can be trained, e.g. *RandomForestRegressor* (it trains many decision trees on subsets of features and averages their preidictions). The use of several models on top of each other: *Ensemble Learning*

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor()
model.fit(train_data_prepared, train_data_labels)
scores = cross_val_score(model, train_data_prepared, train_data_labels,
            scoring="neg_mean_squared_error", cv=10)
model_rmse_scores = np.sqrt(-scores)
```

Before deeping into a model (tweaking hyperparameters), try several different and keep the best ones! Typically keep between 2 and 5 models.
Save:

- models
- hyperparameters
- trained parameters
- cross-validation scores
- actuall predictions

This allows to easily compare the different models.

```
import joblib

joblib.dump(model, 'model.pkl')
# model = joblib.load('model.pkl')
```

2.6 Fine-tune your model

How to fine-tune:

Grid Search

Goes through the hyperparameter space and train several models and evaluate. Once all are trained, one can get the best parameters (or just directly the best model)

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3,10,30],
     'max_features': [2,4,6,8]
    },      # Here: 3x4=12 models to train
    {'bootstrap':[False],
     'n_estimators':[3,10],
     'max_features':[2,3,4]
    },      # here: 2x3=6 models to train with bootstrap on
]

model = RandomForestRegressor()
grid_search = GridSearchCV(mode, param_grid, cv=5,
                scoring='neg_mean_squared_error',
                return_train_score=True)
grid_search.fit(train_data_prepared, train_data_labels)
```

The grid search will do the grid search for each of the dictionaries in `param_grid`.
The results from each:

```
cvres = grid_search.cv_results_
for mean_score, params in zip(["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score, params))
```

Get the best:

```
grid_search.best_params_
grid_search.best_estimator_
```

Normal grid search is fine for a small parameter space, otherwise *RandomizedSearchCV* woule be better. It tries random values of the hyperparameters.

```
n_estimators = [int(x) for x in np.linspace(start=250, stop=300, num=2)]
max_features = ['auto', 'sqrt']
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features}
model = RandomForestRegressor()
model_random = RandomizedSearchCV(estimator=model,
                                  param_distributions=random_grid,
                                  n_iter=1000, cv=2, verbose=2,
                                  random_state=42, n_jobs=-1)
```

Ensemble methods

Combining the best models could perform better → Ensemble

Analyze best models and their errors

Looking at the models gives good insights.
RandomForestRegressor can indicate the relative importance of the attributes:

```
# If grid_search using RandomForestRegressor
feature_importances = grid_search.best_estimator_.featue_importances_

# shown next to the features name:
extra_attribs = ["extra_attribs"]       # Created in the pipeline by combin
# Extract the attributes created from the categorical attributes using OneH
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
# Concatenate all the attributes
attributes = num_attribs + extra_attribs + cat_one_hot_attribs

sorted(zip(feature_importances, attributes), reverse=True)
```

One should look at the specific errors made by the model to try to fix those, example:

- add extra features
- get rid of uninformative features
- cleaning up outliers

Evaluate your system on the test set

Time to try the model on the test set.

- Get the test data and labels.
- run the `full_pipeline`
- call `.transform()`

```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("feature_to_predict", axis=1)
y_test = strat_test_set["feature_to_predict"].copy()

X_test_prepared = full_pipeline.tranform(X_test)
final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

Can also look at the confidence interval:

```
from scipy import stats
confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
```

```
np.sqrt(stats.t.interval(confidence,
                         len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors)))
```

If the results are worse than on the validation set, DO NOT FINETUNE THE HYPERPARAMETERS TO GET BETTER RESULTS!!

2.7 Launch

It is now time to launch, monitor and maintain the system!
Once it is released:
Write monitor code to check the live performance regularly and throw allerts if it drops. Drop in performance may be quick (something break) or slow (model "rot" over time because of the newer data beeing a bit different from the older ones).
The monitoring downstream has to be adapted to the delivered product. → Automate the process as much as possible

- collect fresh data regularly and label it
- Write script to train the model and fine-tune the hyper-parameters (run e.g. everyday)
- Write a script that compares the new and the previous model on the updated test set and deploy the model to production if the performances have not decreased (if it did, why??)

Check the input data quality.
Keep backups of every model! (And have what is necessary to easily move back to a previous model)
Also keep backups of every version of the datasets (in case the newly added data have a lot of outliers).

3 Classification

4 Training models

5 Support Vector Machines

6 Decision Trees

Can perform classification, regression and multiple output tasks.

They are the fundamentals of Random Forests (among the most powerful algo.).

Decision Trees require little data preparation. It does not require feature scaling or centering.

6.1 Training and visualizing

```
data = load_data()
X = data.data
y = data.target
model = DecisionTreeClassifier(max_depth=2)
model.fit(X, y)
```

The tree can be be exported (and then visualized after conversion) using

```
export_graphviz(
    model,
    out_file="output_file.dot"
    feature_names=data.feature_names,
    class_names=data.target_names,
    rounded=True,
    filled=True
)
```

and can be converted in a shell using:

```
dot -Tpng file_name.dot -o file_name.png
```

Reading tree

In the output:

- first line = Condition to be satisfied
- gini = measures the impurity of the node (pure if gini=0) It is pure if all the training instances satisfying the condition belong to the same class.
- samples = how many samples are subject to the condition check (first line)
- value = [.., .., .., ..] = How many training instances of this node applied to
- class = Predicted class

6.2 Making predictions

The gini score for the i^{th} node:

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

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

It can estimate class and probability.
However, the probability is determine by the region (hyper-rectangle) in which it belongs to in the feature space depending on the amount of class of each instances in the hyper-rectangle.

6.3 Making predictions

Scikit-learn uses the Classification and Regression (CART) algorithm to train decision trees and it only allow to split each node into two categories.

There exists other algorithms (ID3) which allow to split into more children categories.

The CART algorithm

Split the training set into sub-set by setting threshold t_k on the feature k.
Which pair (k, t_k)? $\rightarrow$ produces the purest subsets.
Cost function:

$$J\left(k, tk\right) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}} \tag{2}$$

with $G_{l/r}$ measures the impurity of the left/right subset and $m_{l/r}$ the number of instances in the left/right subset.
This method gets itterated at each nodes until reaching the desired depth or if it cannot reduce the impurity.

Gini impurity or Entropy

sklearn $\rightarrow$ Gini impurity (default) but entropy can also be chosen exist.

- thermo.:
  Entropy $\rightarrow$ measures disorder.
- Shannon's information theory:
  It measures the average information content of a message.
  0 = all messages are the same.
- Machine learning: $\rightarrow$ measures impurity. 0 = if set contains instances of only one class.
  Definition: $H_i = -\sum_{k=1}^{n} p_{i,k} log_2(p_{i,k})$

Gini or entropy?
Usually, no big difference.

Gini

- Faster to computer $\rightarrow$ good default
- Tends to isolate the most frequent class in its own branch

Entropy
Produces more balanced trees.

Regularization hyperparameters

Decision tree does not assume anything about the data and just fit the training data. → Often overfitting.

"non-parametric" model because the number of parameters is not set before training. To not overfit → limitation during training = "Regularization".

- `max_depth`
- `min_samples_split`: min number of sample a node must have before it can split
- `min_samples_leaf`: minimum number of samples a leaf must have
- `min_weight_fraction_leaf`: min fraction of instances (compared to total instances) a leaf must have
- `max_leaf_nodes`: max number of leaf nodes
- `max_features`

⇒ regularize the model

6.4 Regression

Trees can also be used for regressions.
Following the fitted tree result in predicting values.
Instead of minimizing the impurity (as in classification), it minimizes the "MSE".
Again, this is prone to overfitting and regularization should be used (e.g. `min_samples_leaf`).

6.5 Limitations

Perpendicular splits

A training set rotated by 45° in the feature space might very well, or badly be splitted using decision tree.
To limit this: ⇒ use Principal Component Analysis.

Sensitivity

Sensitive to small changes in the training set.
Random Forests can limit this by averaging predictions from many trees.

7 Ensemble learning and Random Forest

Having several classifiers → use all of them to predict an output.
This usually gives better prediction than each of them separately

7.1 Voting Classifier

`Hard voring classifier`: selecting the class with most votes
This works best if the classifiers are independent from each other:
⇒ wroks best for different algorithms which makes different types of errors.

```
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
```

If all the classifiers have a `.predict_proba` ⇒ the classification can be made using probabilities (average) using `voting='soft'` for the voting classifier.

7.1.1 Bagging/Pasting

Alternative ways to get a diverse set of predictors is to use the same algorithm for every predictor but to train them on different subsets of the training set. But always picking a number of instances equal to the size of the training set.

Bagging (bootstrap aggregating)
When sampling is performed `with` replacement.
It allows a sample (training instance) to be picked several time for one predictor and for several predictors.

Pasting
When sampling is performed `without` replacement.
Samples can be repicked but only across multiple predictors.

⇒ prediction for a new instance by aggregating all the predictions.
aggregation: `statistical mode`

- = hard voting for classifications.
- = average for regression.

Each predictor has a higher bias but the aggregation reduces varience and bias.

The predictors can be training in parallel and also the predictions.
→ Makes bagging and pasting scale well.
For bagging and pasting using `sklearn`:

```
from sklearn.ensemble import BaggingClassifier
bag_clf = BaggingClassifier(
        DecisionTreeClassifier(), n_estimators=500,
        max_samples=100, bootstrap=True, n_jobs=-1)
```

Bootstraping → more diversity in the subsets ⇒ slightly better bias than pasting.

Often bagging leads to better models. But cross-validation of bagging and pasting can be done too see what works best.

out-of-bag
Because bagging allows for repicking → only about 63% of the instances get used.
The remaining 37% are out-of-bag (and not sampled by that one classifier).
Because the predictor never sees those instances, they can be used for evaluating the expected accuracy on the test set:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    bootstrap=True, n_jobs=-1, oob_score=True)
bag_clf.oob_score_
```

Random Patches and Random Subspaces
`BaggingClassifier` can sample features using `max_features` and `bootstrap_features`.(work like`samples` and `boostrap` but for features).

- **Random Patches:** sampling instances and features
  ⇒ useful when *high-dimensional inputs* (e.g. Images).
- **Random Subspaces:** only sampling features
  ⇒ keeping all training instances ($\max_s amples = 1$, $bootstrap = 0$).

7.2 Random Forests

Random Forest = ensemble of decision trees.
Usually with `max_samples` = size of training set.

Using `RandomForestClassifier` is more optimized for decision trees.

```
rnd_clf = RandomForestClassifier(n_estimators=500,
            max_leaf_nodes=16, n_jobs=-1)
```

Random forest → more randomness:

- at each node → look for the best feature in a random subset of features (instead of all features.)

Extra-Trees Extremely Randomized Trees ensemble
Can make the trees more random by using random thresholds for each feature rather than searching for the best thresholds.
It makes it faster than standard Random Forest.

```
from sklearn.ensemble import ExtraTreesClassifier
et_clf = ExtraTreesClassifier()
```

ExtraTreesRegressor
Similar API to `RandomForestRegressor`.

Feature Importance
By looking at which feature reduces the impurity the most on average (across all trees).

```
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rnd_clf.fit(X, y)
rnd_clf.feature_importances_
```

7.3 Boosting

Boosting

Any ensemble method that combine weak learniners into a strong learner.
Idea:

- Train predictors sequentially, each trying to correct its predecessor.

7.3.1 AdaBoost (Adaptive Boosting)

To be a better predictor than predecessor:
→ Adjust for the instances where the predecessor underfitted.
⇒ Predictors focus more and more on the harder instances.

AdaBoost:

1. trains a base classifier (e.g. Decision Tree) and makes prediction on training set.
2. Increase the relative weight of misclassified training instances
3. Trains a new classifier with updated weights and make new predictions on the training set.
4. etc

Once all predictors are trained → make prediction:

- like bagging/pasting but the predictors have different weights depending on their accuracy on training set

Can only be partially parallelized!

Algorithm

1. Each instance initial weight is set: $w^{(i)} = 1/m$.
2. Train first predictor and calculate the weighted error rate r_1 for this predictor:
$$r_j = \frac{\sum_{i=1, \hat{y}_j^{(i)} \neq y^{(i)}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}} \tag{3}$$
3. Then calculate the predictors weights with η the learning rate:
$$\alpha_i = \eta \log \frac{1 - r_j}{r_j} \tag{4}$$
4. update the weights of the misclassified instances
$$w^{(i)} \leftarrow w^{(i)} \exp(\alpha_j) \text{if} \hat{\text{y}}_{\text{j}}^{(\text{i})} \neq \text{y}^{(\text{i})} \tag{5}$$
otherwise, keep the same weight.

5. Normalize all the weights
6. Train a NEW predictor with the UPDATED weights

Predictions

Take all the predictor weights and take the class whose sum of weights is largest:

$$\hat{y}(\mathbf{x}) = \text{argmax}_k \sum_{j=1, \hat{y}_j(\mathbf{x})=k}^{N} \alpha_j \tag{6}$$

N = number of predictors

Scikit-learn uses a multiclass version of AdaBoost `SAMME` (Stagewise Additive Modeling using a Multiclass Exponential loss function) which is similar to adaboost if only two classes.
If predictors can estimate class probabilities (better than using predictions) `predict_proba()` ⇒ `SAMME.R`

```
fromsklearn.ensemble import AdaBoostClassifier
ada_clf = AdaBoostClassifier(
        DecisionTreeClassifier(max_depth=1), n_estimators=500,
        algorithm="SAMME.R", learning_rate=0.5)
```

7.3.2 Gradient Boosting

Sequentially adding predictors to an ensemble, always correcting the previous ones.
Gradient Boosting tries to fit the new predictor to the residual errors made by the previous predictor.

```
from sklearn.tree import DecisionTreeRegressor
tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)

y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg2.fit(X, y2)

y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg3.fit(X, y3)

# Can make a prediction by adding up the predictions from all trees
y_pred = sum(tree.predict([[0.8]]) for tree in (tree_reg1,
                                        tree_reg2, tree_reg3) )
```

Instead of doing it manually → `GradientBoostingRegressor`.

```
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor(max_depth=2,
                    n_estimators=3, learning_rate=1.)
gbrt.fit(X, y)
```

`learning_rate`: scales the contrib of each tree. If low value → need more trees but will usually generalize better.
The complexity of the trees and how many trees → often underfitting or overfitting.
⇒ possibility is to use: early stop `staged_predict()`.

```
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120,
            random_state=42)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
            for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(max_depth=2,
        n_estimators=bst_n_estimators, random_state=42)
gbrt_best.fit(X_train, y_train)
```

`GradientBoostingRegressor` has `subsample` hyperparameter: to train on a fraction of the training instances.
⇒ **Stochastic Gradient Boosting**
An optimized implementation of Gradient Boosting : `XGBoost`

```
import xgboost
xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

7.4 Stacking (stacked generalization)

Ensemble method.

- Instead of using trivial functions for aggragating the predictors (e.g. hard voting)
  → Training a model to do the aggregation (a `blender` or `meta learner`)
- The `blender` takes the predictions from each predictor and makes a final prediction

1. Training set is split into two subsets
2. The first subset is used to train the predictors

3. The trained predictors are used to make predictions on the second sub-set (`held out set`) (to ensure clean preductions)

4. The predicted values are used as input features and keep their target values.

8 Dimensionality Reduction

In data, often too many features (more than necessary).
→ Can make it harder to predict: `Curse of dimensionality`.

In general reducing dimensionality will result in lower performance but faster training. Rarely (but sometimes), it may get rid of undesirable features and will perform better than without dimensionality reduction.

Dimensionality reduction: useful for data visualization (reducing to 2/3 dimensions → can make plot easily understandable by people.)

Two main approaches:

- Projection
- Manifold Learning

Manifold techniques: PCA, Kernel PCA, and LLE.

Problem of high dimensions:

- Typically, two points are farther apart in higher dimensions than lower dimensions

→ Training instances are sparse and far from each other

→ New instances may be far from training instances ⇒ make worse predictions than in lower dimension (larger extrapolation).

→ The larger the dimensions, the greater the risk of overfitting.

Theoretical solution:

- Increase the size of the training set

In practice, not possible, the number of instances needed grows exponentionally with the dimension

8.1 Main approaches to dimensionality reduction

Projection

- Often the data are not spread out uniformly across dimensions

→ Project onto the subspace.

- Projection not always the best option (bad if subspace is not flat i.e. if it rolls in the higher dimensional space)

⇒ Manifold Learning

Manifold Learning

- d-dimensional manifold in a n-dim. space with the manifold locally a hyper-plane.
- Dimensionality reduction → modeling the manifold on which the training instances are (= Manifold Learning)
- Manifold assumption:
  - real-world high-d datasets lie close to a much lower dimensional manifold.

8.2 PCA

- Most popular
  1. Identifies the hyperplane that lies closest to the data
  2. Project the data onto it
- Idea: Find the subspace (Spectral theorem or more generally: Singular Value Decomposition)
- Each axis is called the i$^{\text{th}}$ principal component
- Project data onto the d-dimensional hyperplane using the relevant principal components:
  $\mathbf{X}_{d-project} = \mathbf{X}\mathbf{W}_d$

```
from sklearn.decomposition import PCA
# Does not require centering when using sklearn
X = load_data()
# Keep the 2 components corresponding to largest singular (eigenvalue)
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
# Contains the basis vector of the subspace
print(pca.components_)
# proportion of dataset's variance along those axis:
print(pca.explained_variance_)
```

number of dimensions

- Until the variance adds up to a certain % (e.g. 95%).
- Or, if for viz: 2, 3, max 4

To reach a certain variance, several ways:

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum>=0.95) + 1
pca = PCA(n_components = d)     # Should be part of the pipeline
X_reduced = pca.fit_transform(X_train)
# We can "get back the data" (with a bit of loss of course)
# by doing the inverse transfo.
X_recovered = pca.inverse_transform(X_reduced)
```

or more simply:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

To speed it up, one can use randomized PCA which search for approximate vectors:

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

This implementation requires the whole dataset, if not possible (not enough memory or online training), use incremental PCA:

→ use mini-batches instead of the full data set

⇒ use `partial_fit`

```
from sklearn.decomposition import IncrementalPCA
n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

8.3 Kernel PCA (kPCA)

Similar to the kernel trick used for SVM.
Allows, nonlinear projections for dimensionality reduction.
Good at:

- preserving clusters of instances after projection
- unrolling datasets lying in a twisted manifold.

Unsupervised ML:

→ no clear performance measure

- If used for classification →grid search for kernel and hyperparameters that gives best perf. on classification

```
clf = Pipeline([
    ("kpca", KernelPCA(n_components=2)),
    ("log_reg", LogisticRegression())
    ])

param_grid = [{
    "kpca__gamma": np.linspace(0.03, 0.05, 10),
    "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
print(grid_search.best_params_)
```

- If really unsupervised: →hyperparameters that minimize the reconstruction error

→ set `fit_inverse_transform=True`

```
rbf_pca = KernelPCA(n_components=2, kernel="rbf", gamma=0.0433, fit_inv
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
print(mean_squared_error(X, X_preimage))
```

⇒ can now use grid-search with cross-validation to find the kernel and hyperparameters

8.4 LLE

- Locally Linear Embedded
- uses manifold learning i.e. does not rely on projections
- How it works:
  - measures how each training instances are linearly related to its closest neighbour (`n_neighbors`)
  - finds a low-dimensional representation of the training set where the local relationships are satisfied
- Good at unrolling manifold if not too much noise
- How it works: