ML methods (theory)

October 18, 2021

1 Supervised Learning

These notes will go into detail on supervised learning. Most of what we have been doing in Python so far falls under supervised learning.

Supervised learning is machine learning where the algorithm tries to fit a target using a given input. The algorithm learns a rule that it uses to predict the labels for new observations.

The two types of supervised learning are:

- regression
- and classification.

In **regression** supervisd learning, the algorithm tries to predict the value of an output based on inputs.

In **classification** supervised learning, the algorithm tries to identify the category that a set of data items belongs to. These are probability-based, i.e. the outcome is the category to which the model assigns the highest probability of belonging.

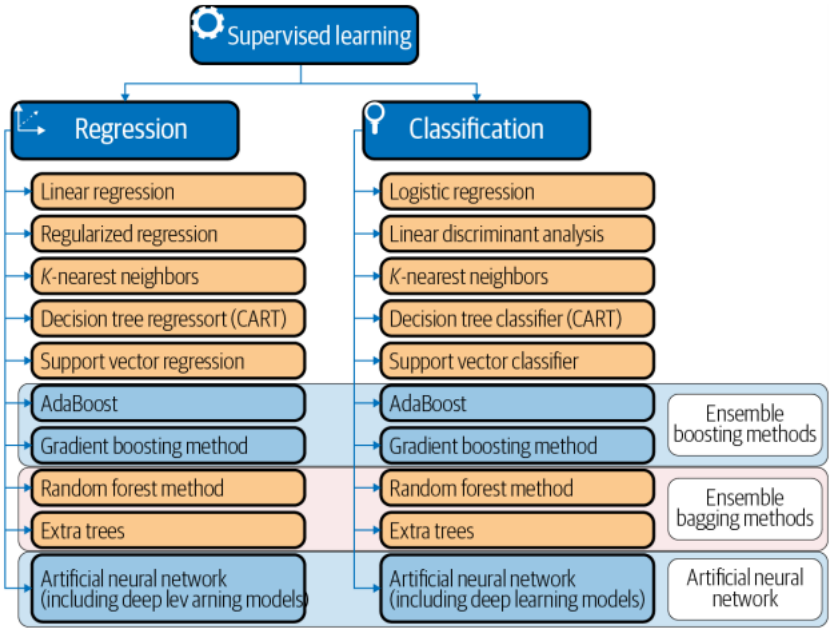
In finance, supervised learning models are widely used for algorithmic trading and asset pricing models. Classification models are widely used in areas of finance where we want to predict a categorical response. Things like Buy/Sell recommendations and fraud detection are great applications for classification algorithms.

1.1 Overview

The following figure breaks down which algorithms can be used for regression problems, classification problems, or both.

Note that there is a significant overlap between the two models. Hence, the models for classification and regression are presented together in this note. The following figure summarizes the list of the models commonly used for classification and regression.

Some models can be used for **both** classification and regression with small modifications. These are K-nearest neighbors, decision trees, support vector, ensemble bagging/boosting methods, and ANNs (including deep neural networks), as shown in the figure below. However, some models, such as linear regression and logistic regression, cannot (or cannot easily) be used for both problem types.



1.2 Linear regression

Linear regression (Ordinary Least Squares Regression or OLS Regression) is the moest well-known algorithm in statistics and machine learning.

Linear regressions are linear models, meaning the model **assumes** a linear relationship between the input variables (x) and one output variable (y).

The model is a function that predicts y given $x_1, x_2, ...x_i$:

$$y = \beta_0 + \beta_1 x_1 + ... + \beta_i x_i$$

where β_0 is the intercept and $\beta_1, ..., \beta_i$ are the coefficients of the regression.

1.2.1 Linear Regression in python

NOTE: all of the following code in these notes is for purposes of **pointing towards the correct tools and won't actually produce anything**.

[1]:
```
# If we want to use linear regressions in Python, we can use

from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

1.2.2 Training a supervised learning model

In this section, we cover the training of a linear regression model. However, the overall concepts and related approaches are relevant to all other supervised learning models.

what is a Cost function? Cost function measures how **inaccurate** the model's predictions are. The *sum of squared residuals (RSS)* as defined below measures the squared sum of the difference between the actual and predicted value and is the cost function for linear regression.

$$RSS = \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{n}\beta_j x_{ij})^2$$

Minimizing the loss function We want to find the parameters that minimize the loss function. For a linear regression, we want to find the values of β (i.e., $\beta_0, \beta_1, ..., \beta_n$) such the difference between each real data point, y, and the model's prediction, $\hat{y}$, is as small as possible.

1.2.3 Grid search

The idea of the grid search is to create a grid of **all possible hyperparameter combinations** and *train the model using each one of them.*

Hyperparameters are the external characteristic of the model, can be considered the model's settings, and are not estimated based on data-like model parameters. These hyperparameters are tuned during grid search to achieve better model performance. Due to its exhaustive search, a grid search is guaranteed to find the optimal parameter within the grid. The cost is that the size of the grid grows exponentially with the addition of more parameters or more considered values.

The **GridSearchCV** class in the **model_selection** module of the **sklearn** package facilitates the systematic evaluation of all combinations of the hyperparameter values that we would like to test.

The *first step* is to create a model object. We then define a dictionary where the keywords name the hyperparameters and the values list the parameter settings to be tested. For example, for linear regression, the hyperparameter is **fit_intercept**, which is a boolean variable that determines whether or not to calculate the intercept for this model. If set to False, no intercept will be used in calculations.

The *second step* is to apply the GridSearchCV object and provide the estimator object and parameter grid, as well as a scoring method and cross validation choice, to the initialization method. Cross validation is a resampling procedure used to evaluate ML models, and scoring parameter is the evaluation metrics of the model.

An example would look something like this:

[2]:
```
from sklearn.linear_model import LinearRegression

from sklearn.model_selection import GridSearchCV
```

```
model = LinearRegression()

# We are creating a variable to represent whether the intercept
# should be calculated or not. If we had other variables that
# we wanted to create, we could store them also in param_grid.
param_grid = {'fit_intercept': [True, False]}

# Now, we are doing a grid search, and specifying the estimator object,
# parameter grid, and a scoring method (we are looking for the best R-squared
 ↪here)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring= 'r2')

# With all settings in place, we can fit GridSearchCV:
grid_result = grid.fit(X, Y)
```

1.2.4 Pros/cons of Linear Regression

Linear regressions are easy to understand and provide results that are easy to intepret.

However, linear regression is prone to overfitting (more on this later) and when a large number of features are present, it may not handle irrelevant features well. Also, if the real relationship between the input and output variables is nonlinear, then linear regression might not perform well.

1.3 Regularized regression

When a linear regression model contains many independent variables, their coefficients will be poorly determined, and the model will have a tendency to fit extremely well to the training data (data used to build the model) but fit poorly to testing data (data used to test how good the model is). This is known as **overfitting** or high variance.

One way to fix overfitting is **regularization**, which involves adding a penalty term to the error or loss function to prevent the coefficients from becoming very large.

So regularization, in simple terms, is a penalty mechanism driving model parameters closer to zero in order to build a model with higher prediction accuracy and interpretation.

It has two advantages:

Prediction accuracy A model with too many parameters might try to fit noise specific to the training data. By shrinking some coefficients close to zero, we can achieve better performance **because the model won't try to fit irrelevant 'noise' that is in the training data into new data**.

Interpretation The smaller the number of predictors/variables, the **easier** it is to interpret what the results mean for the question at hand.

There are 3 common ways to **regularize** a linear regression model:

1. Lasso regression (L1) In a lasso regression, the cost function is RSS (see above) plus a factor of the sum of the absolute value of coefficients.

$$CostFunction = RSS + \lambda * \sum_{j=1}^{p} |\beta_j|$$

This can lead to coefficients of zero, meaning some variables are completely ignored when computing the output. The **larger** the value of λ, the more variables are shrunk to zero. A λ value of zero makes the regression just a basic linear regression.

Note: Since the Lasso regression can eliminate many variables, it can actually help perform **feature selection**, the process of determining which variables are most important for the model.

To run a Lasso regression in Python, we can use scikit-learn.

[1]:
```
from sklearn.linear_model import Lasso
model = Lasso()
model.fit(X, Y)
```

2. Ridge regression (L2) In a ridge regression, the cost function is RSS (see above) plus a factor of the sum of the square of coefficients.

$$CostFunction = RSS + \lambda * \sum_{j=1}^{p} \beta_j^2$$

A ridge regression helps decrease the complexity of a model, but does not reduce the number of variables; it simply shrinks their effect. In other words, ridge regression shrinks the coefficients and helps to reduce the model complexity. Therefore, ridge regression decreases the complexity of a model but does **not** reduce the number of variables; it just shrinks their effect. When λ is closer to zero, the cost function becomes similar to the linear regression cost function. So the lower the constraint (low λ) on the features, the more the model will resemble the linear regression model.

To run a ridge regression in Python, we can use scikit-learn.

[4]:
```
from sklearn.linear_model import Ridge
model = Ridge()
model.fit(X, Y)
```

3. Elastic net Elastic nets are a combination of L1 and L2 regulariztion. That is, Elastic nets add regularization terms to the model, which are a combination of both L1 and L2 regularization. The cost function is as follows:

$$CostFunction = RSS + \lambda * \left((1-\alpha) * \sum_{j=1}^{p} \beta_j^2 + \alpha * \sum_{j=1}^{p} |\beta_j|\right)$$

Here, in addition to λ, we choose an α between 0 and 1.

Therefore, in addition to setting and choosing a λ value, an elastic net also allows us to tune the alpha parameter, where $\alpha = 0$ corresponds to ridge and $\alpha = 1$ to lasso. As a result, we can choose an α value between 0 and 1 to optimize the elastic net. **Effectively, this will shrink some coefficients and set some to 0 for sparse selection**.

We can construct an elastic net in Python using scikit-learn.

[5]:
```
from sklearn.linear_model import ElasticNet
model = ElasticNet
model.fit(X, Y)
```

1.3.1 Logistic Regression

Logistic regressions, also known as logit models, are one of the most well-known algorithms for **classification**. If we train a linear regression on data where $Y = 0$ or 1, then we might end up predicting some probabilities that are either less than zero or greater and one, which doesn't make sense as probabilities.

Instead, we use a **logistic regression model (or logit model)**, which is a modification of linear regression that makes sure to output a probability between zero and one by applying the **sigmoid function**. As in the following equation, similar to linear regression, input values (x) are combined linearly using weights or coefficient values to predict an output value (y). The output coming from the below equation is a **probability** that is transformed into a binary value (0 or 1) to get the model prediction

$$y = \frac{exp(\beta_0 + \beta_1 x_1 + ... + \beta_i x_i)}{1 + exp(\beta_0 + \beta_1 x_1 + ... + \beta_i x_i)}$$

For logistic regressions, the cost function is a measure of how often the model predicted 1 when the real value was 0, or vice versa. We can estimate the coefficients using maximum likelihood estimation (MLE), a method where we choose the model for which the observed data was the most probable.

For more on MLE see https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

We can construct a logistic regression in Python using scikit-learn.

[6]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X, Y)
```

Hyperparameters

Regularization ("penalty" in sklearn) Similar to linear regression, Logistic regressions can also have L1, L2, or elastic net regularizations just like linear regressions. If you're using sklearn, the values in the library are [l1, l2, or elasticnet].

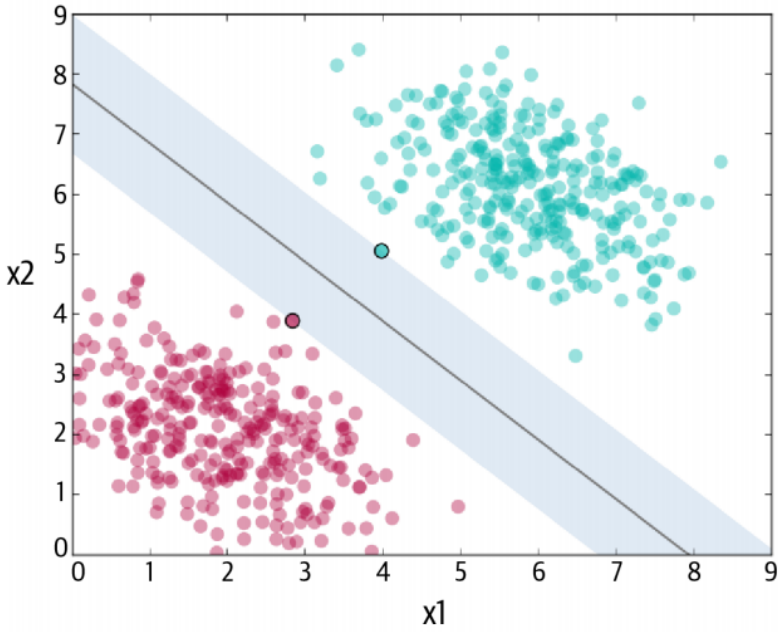
Regularization strength (C in sklearn) This parameter controls regularization strength. Some good values of the penalty parameters are [100, 10, 1.0, 0.1, 0.01].

Pros and Cons The logistic regression is easy to implement and somewhat easy to interpret. The model has small number of hyperparameters. Although there may be risk of overfitting, this may be addressed using L1/L2 regularization, similar to the way we addressed overfitting for the linear regression models.

However, logistic regression can only learn linear functions, so if the relationship between inputs and outputs is much more complex, the model may not perform well. Also, it may not handle irrelevant features well, especially if the features are strongly correlated.

1.3.2 Support Vector Machine

The objective of the **support vector machine (SVM)** algorithm is to **maximize the margin** (shown as shaded area in the following figure), which is defined as the distance between the separating hyperplane (or decision boundary) and the training samples that are closest to this hyperplane, the so-called support vectors. The margin is calculated as the perpendicular distance from the line to only the closest points, as shown in the following figure. Hence, SVM calculates a maximum-margin boundary that leads to a homogeneous partition of all data points.



Obviously, in practice, the data cannot always be perfectly separated like in the figure above. We use a tuning parameter, C, that defines **the magnitude of the wiggle allowed across all dimensions**. The larger the value for C, the more violations of the decision boundary are permitted.

In some cases, it is not possible to find a hyperplane or a linear decision boundary, and kernels are used. A kernel is just a transformation of the input data that allows the SVM algorithm

to treat/process the data more easily. Using kernels, the original data is projected into a higher dimension to classify the data better.

SVMs can also be used for regression problems, where instead the model tries to fits as many observations as possible within the shaded area, with the width of the street (shaded area in the above figure) controlled by a hyperparameter.

Both regression and classification SVMs are easy to implement with scikit-learn:

[7]:
```
#Regression
from sklearn.svm import SVR
model = SVR()
model.fit(X, Y)

#Classification
from sklearn.svm import SVC
model = SVC()
model.fit(X, Y)
```

Hyperparameters The following key parameters are present in the sklearn implementation of SVM and can be tweaked while performing the grid search:

Kernels ("kernel" in sklearn) The choice of kernel controls how the input variables will be projected. *Linear* and *RBF* are the most popoular.

For more on RBF see: https://en.wikipedia.org/wiki/Radial_basis_function_kernel

Penalty (C in sklearn) The penalty parameter tells the SVM optimization **how much we want to avoid misclassifying** each training example. If the penalty parameter is large, the optimization will choose a smaller-margin decision boundary. Good values might be a log scale from 10 to 1,000.

Pros and Cons SVMs are generally good models to avoid overfitting, and they can also handle nonlinear relationships well.

However, SVM can be very memory-intensive and computationally inefficient. Also, its performance declines with large datasets, it requires feature scaling, and the hyperparamters can be hard to interpret.

1.3.3 K-Nearest Neighbors (KNN)

In KNN models, predictions are made by searching through the entire training set for the K most similar observations (neighbors), and summarizing the output variable for those K instances.

To identify which of the observations in the training data are most similar to a new input, a distance measure is used. Some popular distances are the Euclidean distance:

$$d(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

and the Manhattan distance:

$$d(a,b) = \sum_{i=1}^{n}|a_i - b_i|$$

Generally, the Euclidean distance is a better measure if the input variables are similar in type, and the Manhattan distance is a better measure if the input variables are not similar in type.

The steps of KNN are as follows:

1. Choose the number of K, and a distance metric
2. Find the K-nearest neighbors of the data point that we want to classify
3. Assign the class label by a majority vote (or a regression value by a mean value)

KNN regression and classification models can be constructed using the sklearn package of Python, as shown in the following code:

[8]:
```
# Classification
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier()
model.fit(X, Y)

# Regression
from sklearn.neighbors import KNeighborsRegressor
model = KNeighborsRegressor()
model.fit(X, Y)
```

Hyperparameters The following key parameters are present in the sklearn implementation of KNN and can be tweaked while performing the grid search:

Number of neighbors ("n_neighbors" in sklearn) The most important hyperparameter for KNN is the **number of neighbors** (n_neighbors). Some good values are in the range of 1 to 20.

Distance metric ("metric" in sklearn) We need to choose our distance metric, e.g. *euclidean* or *manhattan*.

Pros and Cons In terms of advantages, no training is involved and hence there is no learning phase. Since the algorithm requires no training before making predictions, new data can be added seamlessly without impacting the accuracy of the algorithm. It is intuitive and easy to understand. The model naturally handles multiclass classification and can learn complex decision boundaries.

KNN is effective if the training data is large. It is also robust to noisy data, and there is no need to filter the outliers.

However, the distance metric can be difficult to choose. Also, KNN performs poorly on high dimensional datasets. It is expensive and slow to predict new instances because the distance to all neighbors must be recalculated. KNN is sensitive to noise in the dataset. We need to manually input missing values and remove outliers.

1.3.4 Linear Disciminant Analysis

In Linear Discriminant Analysis, the data is projected onto a **lower-dimensional space** in a way that the class **separability is maximized** and the **variance within a class is minimized**.

The following assumptions about the data are required:

1. Data is normally distributed (i.e. each variable follows a bell curve distribution)
2. Each attribute has the same variance (i.e. the values of each variable vary around the mean by the same amount on average)

When the LDA is trained, the mean and covariance matrix of each class are computed. To make a prediction, LDA esimates the probability that a new set of input belongs to every class. The output class is the one with the highest probability.

We can use scikit-learn in python to use LDA:

[9]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
model = LinearDiscriminantAnalysis()
model.fit(X, Y)
```

Hyperparameters The hyperparameter that we must choose for LDA is the number of components for dimensionality reduction. This is "n_components" in scikit-learn.

Pros and Cons LDA is simple and easy to implement.

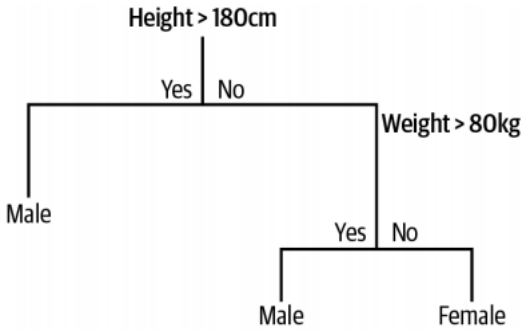
However, it requires feature scaling and involves complex matrix operations.

1.3.5 Classification and Regression Trees (CART) – Decision Trees

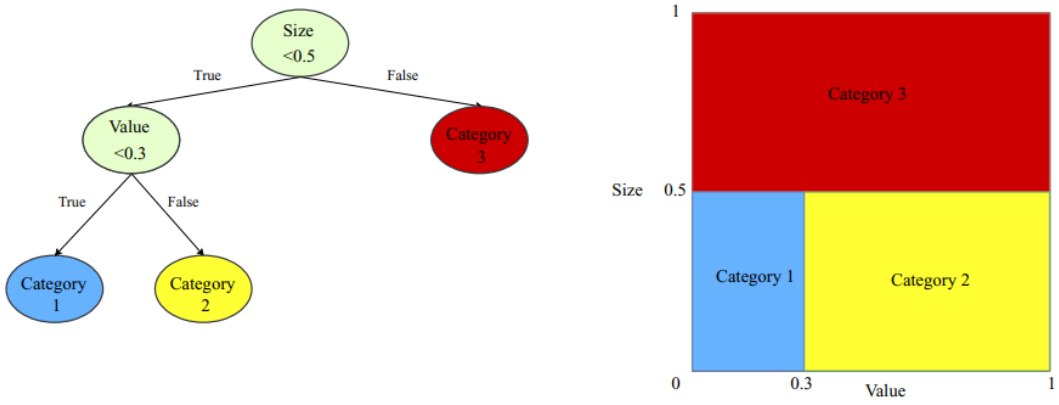
Decision tree classifiers, also referred to as CART, **are similar to asking the data a series of questions and following the answers down respective branches**.

In the most general terms, the purpose of an analysis via tree-building algorithms is to determine a set of *if–then logical (split)* conditions that permit accurate prediction or classification of cases. Classification and regression trees (or CART or decision tree classifiers) are attractive models if we care about interpretability.

The model can be represented by a binary tree (or decision tree), where each node is an input variable x with a split point and each leaf contains an output variable y for prediction.



The Review of Financial Studies / v 33 n 5 2020



Learning a CART model Creating a binary tree is actually a process of dividing up the input space. A greedy approach called *recursive binary splitting* is used to divide the space. This is a numerical procedure in which all the values are lined up and different split points are tried and tested using a cost (loss) function.

The split with the best cost (lowest cost, because we minimize cost) is selected. All input variables and all possible split points are evaluated and chosen in a greedy manner (e.g., the very best split point is chosen each time).

- For **regression** predictive modeling problems, the cost function that is minimized to choose split points is the sum of squared errors across all training samples that fall within the rectangle:

$$Lossfunction : \sum_{i=1}^{n}(y_i - prediction_i)^2$$

where y_i is the output for the trianing sample and $prediction_i$ is the output for the rectangle.

- For **classification**, the **Gini cost function** is used; it provides an indication of how pure the leaf nodes are (i.e., how mixed the training data assigned to each node is) and is defined as:

$$G = \sum_{i=1}^{n} p_k * (1 - p_k)$$

where G is the Gini cost over all classes and p_k is the number of training instances with class k in the rectangle of interest. A node that has all classes of the same type (perfect class purity) will have $G = 0$, while a node that has a 50–50 split of classes for a binary classification problem (worst purity) will have $G = 0.5$.

Stopping criterion The recursive binary splitting procedure needs to know when to stop splitting as it works its way down the tree with the training data. The most common stopping procedure is to use a minimum count on the number of training instances assigned to each leaf node. If the count is less than some minimum, then the split is not accepted and the node is taken as a final leaf node.

Pruning the tree The stopping criterion is important as it strongly influences model performance. The complexity of a decision tree is the number of splits in the tree. **Simpler trees are computationally faster, less prone to overfitting, and easier to understand**. One way to prune is to work through each leaf node in the tree and evaluate the effect of removing it using a test set. A leaf node is only remooved if doing so results in a drop in the overall clost function on the entire test set.

In Python, we can use scikit-learn to build decision trees:

[10]:
```
# Classification
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(X, Y)

# Regression
from sklearn.tree import DecisionTreeRegressor
model = DecisionTreeRegressor()
model.fit(X, Y)
```

Hyperparameters The most important hyperparamter in CART (decision tree) is the **maximum depth of the tree model**, which is '**max_depth**' in scikit-learn. Some good values for the maximum depth are between 2 and 30, depending on the number of input variables in the data.

Pros and cons Decision trees are easy to understand, and can learn complex relationships between variables. The data typically don't need to be scaled, and feature importance is built-in to the model due to the way the nodes are built. It also performs well on large datasets.

However, unless pruning is used, CART is likely to overfit to the training data. It can be very nonrobust, meaning that small changes in the training dataset can lead to quite major differences in the hypothesis function that gets learned. CART (decision tree) generally has worse performance than ensemble models, which are covered next.

1.3.6 Ensemble Models

The goal of ensemble models is to combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone. For example, assuming that we collected predictions from 10 experts, ensemble methods would allow us to strategically combine their predictions to come up with a prediction that is more accurate and robust than the experts' individual predictions.

The two most popular ensemble methods are:

- Bagging and
- Boosting.

Bagging, also known as *boostrap aggregation*, is the training of several individual models in a **parallel way**. Each model is trained by a random subset of the data.

Boosting is an ensemble technique of training several individual models in a **sequential way**. A model is built from the trainng data and then a second model is created that attempts to correct the errors of the first model. Each individual model learns from mistakes made by the previous model.

By combining individual models, the ensemble model tends to be more flexible (less bias) and less data-sensitive (less variance). Ensemble methods combine multiple, simpler algorithms to obtain better performance.

We next cover random forest, AdaBoost, the gradient boosting method, and extra trees, along with their implementation using sklearn package.

1.3.7 Random forests

A random forest is a **tweaked verseion of bagged decision trees**. The steps to bagging are as follows:

1. Create many (for example, 100) random subsamples of our dataset.
2. Train a CART model on each sample.
3. Given a new dataset, calculate the average prediction from each model and aggregate the prediction by each tree to assign the final label by a majority vote.

The problem with decision trees is that, even after bagging, each of the underlying decision trees can have a lot of similairities which results in high correlation in their predictions. Combining predictions works better if the predictions from the underlying submodels are uncorrelated or very weakly correlated. This is the idea behind random forests.

In CART, when selecting a split point, the learning algorithm is allowed to look through all variables and all variable values in order to select the most optimal split point. The random forest algorithm changes this procedure such that each subtree can access only a random sample of features when selecting the split points. The number of features that can be searched at each split point (m) must be specified as a parameter to the algorithm.

As the bagged decision trees are constructed, we can calculate how much the error function drops for a variable at each split point. In regression problems, this may be the drop in sum squared error, and in classification, this might be the Gini cost. The bagged method can provide feature importance by calculating and averaging the error function drop for individual variables.

We can use scikit-learn to build random forests.

[11]:
```
# Classification
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(X, Y)

#Regression
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor()
model.fit(X, Y)
```

Hyperparameters Some of the main hyperparameters that are present in the sklearn implementation of random forest and that can be tweaked while performing the grid search are:

Maximum number of features ('max_features' in sklearn) This is the most important parameter. It is the number of random features to sample at each split point. You could try a range of integer values, such as 1 to 20, or 1 to half the number of input features.

Number of estimators ('n_estimators' in sklearn) This parameter represents the number of trees. Ideally, this should be increased until no further improvement is seen in the model. Good values might be a log scale from 10 to 1,000.

Pros and cons The random forest algorithm (or model) has gained huge popularity in ML applications during the last decade due to its good performance, scalability, and ease of use. It scales to large datasets and is generally robust to overfitting. The algorithm doesn't need the data to be scaled and can model a nonlinear relationship.

However, the results can sometimes be difficiult to intepret, and they also can't predict beyond the range in the training data and may overfit datasets that are particularly noisy.

1.3.8 Extra trees

Extra trees, otherwise known as extremely randomized trees, is a variant of a random forest; it builds multiple trees and splits nodes using random subsets of features similar to random forest. However, unlike random forest, where observations are drawn with replacement, the observations are drawn without replacement in extra trees. So there is no repetition of observations.

Additionally, random forest selects the best split to convert the parent into the two most homogeneous child nodes. However, extra trees selects a random split to divide the parent node into two random child nodes. In extra trees, randomness doesn't come from bootstrapping the data; it

comes from the random splits of all observations. The advantages and disadvantages of extra trees are similar to those of random forest.

In real-world cases, performance is comparable to an ordinary random forest, some- times a bit better.

The hyperparameters of extra trees are similar to random forest, as shown in the pre- vious section.

It can be implemented using scikit-learn.

[12]:
```
# Classification
from sklearn.ensemble import ExtraTreesClassifier
model = ExtraTreesClassifier()
model.fit(X, Y)

#Regression
from sklearn.ensemble import ExtraTreesRegressor
model = ExtraTreesRegressor()
model.fit(X, Y)
```

1.3.9 Adaptive Boosting (AdaBoost)

Adaptive Boosting or AdaBoost is a boosting technique in which the basic idea is to try predictors **sequentially**, and each subsequent model attempts to fix the errors of its predecessor. At each iteration, the AdaBoost algorithm changes the sample distribution by modifying the weights attached to each of the instances. It increases the weights of the wrongly predicted instances and decreases the ones of the correctly predicted instances.

The steps of AdaBoost are as follows:

1. Initially, all observations are given equal weights.
2. A model is built on a subset of data, and using this model, predictions are made on the whole dataset. Errors are calculated by comparing the predictions and actual values.
3. While creating the next model, higher weights are given to the data points that were predicted incorrectly. Weights can be determined using the error value. For instance, the higher the error, the more weight is assigned to the observation.
4. This process is repeated until the error function does not change, or until the maximum limit of the number of estimators is reached.

We can use scikit-learn for AdaBoost:

[13]:
```
# Classification
from sklearn.ensemble import AdaBoostClassifier
model = AdaBoostClassifier()
model.fit(X, Y)

# Regression
from sklearn.ensemble import AdaBoostRegressor
```

```
model = AdaBoostRegressor()
model.fit(X, Y)
```

Hyperparameters Some of the main hyperparameters that are present in the sklearn implementation of AdaBoost and that can be tweaked while performing the grid search are as follows:

Learning rate ('learning_rate' in sklearn) Learning rate shrinks the contribution of each classifier/regressor. It can be con- sidered on a log scale. The sample values for grid search can be 0.001, 0.01, and 0.1.

Number of estimators ('n_estimators' in sklearn) This parameter represents the number of trees. Ideally, this should be increased until no further improvement is seen in the model. Good values might be a log scale from 10 to 1,000.

Pros and cons AdaBoost has a high degree of precision; it can achieve similar performance to other models with much less tweaking. It can also handle nonlinear relationships and unscaled data.

However, traning AdaBoost algorithms can take a while, and can be sensitive to noisy data, unbalanced data, and outliers in the data.

1.3.10 Gradient boosting method

Gradient boosting is another technique similar to AdaBoost, where the idea is to try to predictors **sequentially**. Gradient boosting works by sequentially adding the previous underfitted predictions to the ensemble, ensuring the errors made previously are corrected.

The steps of gradient boosting are as follows:

1. A model (which can be referred to as the first weak learner) is built on a subset of data. Using this model, predictions are made on the whole dataset.
2. Errors are calculated by comparing the predictions and actual values, and the loss is calculated using the loss function.
3. A new model is created using the errors of the previous step as the target variable. The objective is to find the best split in the data to minimize the error. The predictions made by this new model are combined with the predictions of the previous. New errors are calculated using this predicted value and actual value.
4. This process is repeated until the error function does not change or until the maximum limit of the number of estimators is reached.

Contrary to AdaBoost, which tweaks the instance weights at every interaction, this method tries to fit the new predictor to the residual errors made by the previous predictor.

We can use scikit-learn to implement gradient boosting:

[14]:
```
# Classification
from sklearn.ensemble import GradientBoostingClassifier
model = GradientBoostingClassifier()
model.fit(X, Y)

# Regression
from sklearn.ensemble import GradientBoostingRegressor
model = GradientBoostingRegressor()
model.fit(X, Y)
```

Pros and cons **Gradient boosting is a good technique for datasets with a lot of missing data, highly correlated variables, and irrelevant variables.**

It can handle unscaled data, nonlinear relationships, and naturally assigns feature importance scores.

However, it can be more prone to *overfitting* than random forests, has many hyperparameters, and feature importance may vary a lot with the training dataset.

1.3.11 ANN Models

Previously, we implemented basic artificial neural networks (ANN) in Python. This section will go into more details behind ANN techniques.

Neural networks are reducible to a classification or regression model with the activation function of the node in the output layer.

In the case of a regression problem, the output node has linear activation function (or no activation function). A linear func- tion produces a continuous output ranging from -inf to +inf. Hence, the output layer will be the linear function of the nodes in the layer before the output layer, and it will be a regression-based model.

In the case of a classification problem, the output node has a sigmoid or softmax acti- vation function. A sigmoid or softmax function produces an output ranging from zero to one to represent the probability of target value. Softmax function can also be used for multiple groups for classification.

We can construct ANN models with scikit-learn:

[15]:
```
# Classification
from sklearn.neural_network import MLPClassifier
model = MLPClassifier()
model.fit(X, Y)


# Regression
from sklearn.neural_network import MLPRegressor
model = MLPRegressor()
model.fit(X, Y)
```

Hyperparameters

Hidden Layers ('hidden_layer_sizes' in sklearn) It represents the number of layers and nodes in the ANN architecture. In sklearn implementation of ANN, the ith element represents the number of neurons in the ith hidden layer. A sample value for grid search in the sklearn implementation can be [(20,), (50,), (20, 20), (20, 30, 20)].

Activation Function ('activation' in sklearn) This is the activation function of a hidden later. Some of the most common functions are *sigmoid*, *relu*, and *tanh*.

Deep neural networks ANNs with more than one hidden layer are caled 'deep' networks. We use the library Keras to build such networks. The main downside to using ANNs is that their results are very difficult to interpret. They can also be computationally expensive to train. ANNs are very flexible however, and are a useful tool to model complex datasets, on which more simple models fail to perform well.

1.4 Model performance

In the previous section, we discussed grid search as a way to find the right hyperparameter to achieve better performance. In this section, we will expand on that process by discussing the key components of evaluating the model performance, which are overfitting, cross validation, and evaluation metrics.

1.4.1 Overfitting and Underfitting

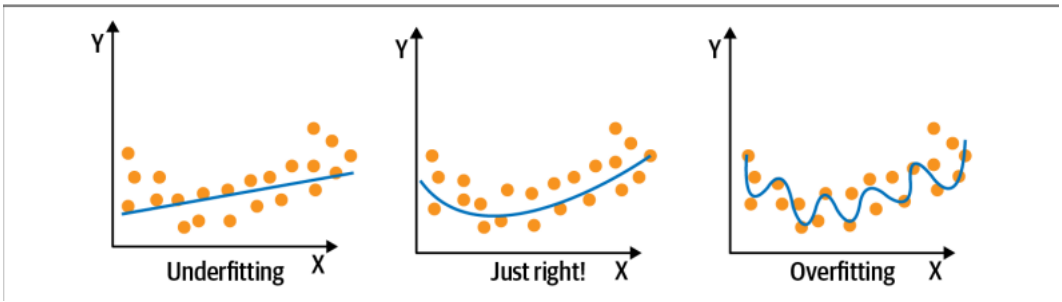
The most problem common in machine learning, which we have already discussed, is **overfitting**, which occurs when a model explains the training data perfectly, but does not generalize well to unseen test data.

Overfitting is a reult of the model picking up noise in the data that aren't representative of the true patterns in the data.

Underfitting, on the other hand, is when the model is not complex enough to be representative of the true pattern we are trying to discover.

The concepts of overfitting and underfitting are closely linked to bias-variance tradeoff. Bias refers to the error due to overly simplistic assumptions or faulty assumptions in the learning algorithm. Bias results in underfitting of the data, as shown in the left-hand panel of the following figure. A high bias means our learning algorithm is missing important trends among the features. Variance refers to the error due to an overly complex model that tries to fit the training data as closely as possible. In high variance cases, the model's predicted values are extremely close to the actual values from the training set. High variance gives rise to overfitting, as shown in the right-hand panel of the following figure. Ultimately, in order to have a good model, we need low bias and low variance.

We can combat overfitting by using ***more training data*** *and using* ***regularization****.*



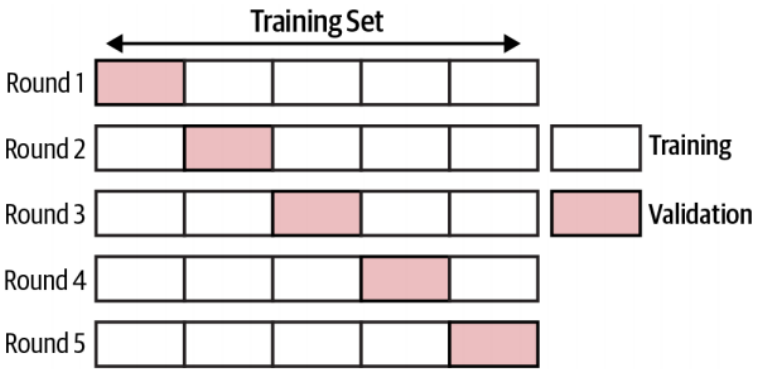
1.4.2 Cross Validation

One of the challenges of machine learning is training models that are able to general- ize well to unseen data (overfitting versus underfitting or a bias-variance trade-off). The main idea behind cross validation is to split the data one time or several times so that each split is used once as a validation set and the remainder is used as a training set: part of the data (the training sample) is used to train the algorithm, and the remaining part (the validation sample) is used for estimating the risk of the algorithm. Cross validation allows us to obtain reliable estimates of the model's generalization error.

In k-fold cross validation, we randomly split the training data into k folds. Then, the model is trained using k - 1 folds and performance is evaluated on the kth fold. We repeat this k times and average the scores.

The following figure shows an example of cross validation, where the data is split into 5 sets and in each round one of the sets is used for validation.

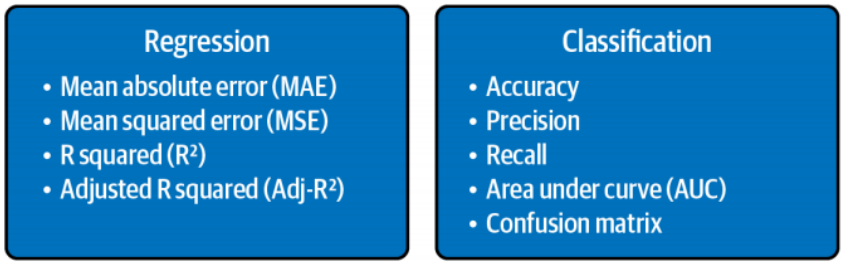
A potential drawback of cross validation is that cross validation can be computationally expensive, especially when paired with a grid search for hyperparameter tuning.



1.4.3 Evaluation Metrics

The metrics used to evaluate the machine learning algorithms are very important. The choice of metrics to use influences how the performance of machine learning algorithms is measured and compared. The metrics influence both how you weight the importance of different characteristics in the results and your ultimate choice of algorithm.

The following figure summarizes the main evaluation metrics for regression and classification.



Let us look at the evaluation metrics for supervised regression.

Mean absolute error The MAE is the sum of the **absolute differences** btween predictions and actual values. The MAE is a linear score, which means that all the individual differences are weighted equally in the average. This gives an idea of the magnitude of the error, but does not indicate the direction.

Mean squared error The MSE is **the sample standard deviation of the differences** between predicted values and observed values (called residuals). This is similar to the to the MAE in that it give an idea of the magnitude of the error. The square root of the MSE converts the units back to the original units of the output variable and can be meaningful for interpretation. This measure is the *root mean squared error* (RMSE).

R^2 **metric** The R^2 metric gives an indication of the **goodness-of-fit** of the preictions to the actual values. This is a number between zero and one; zero represents no fit and one represents a perfect fit.

Adjusted R^2 metric Just like R^2, adjusted R^2 also shows how well terms fit a curve or line but adjusts for the number of terms in a model. It is given in the following formula:

$$R^2_{adj} = 1 - \left(\frac{(1 - R^2)(n - 1)}{n - k - 1} \right)$$

where n is the total number of observations and k is the number of predictors. The adjusted-R^2 is always less than or equal to R^2.

Choosing a metric In terms of a preference among these evaluation metrics, if the main goal is predictive accuracy, then RMSE is best. It is computationally simple and is easily differentiable. The loss is symmetric, but larger errors weigh more in the calculation. The MAEs are symmetric but do not weigh larger errors more. R^2 and adjusted R^2 are often used for explanatory purposes by indicating how well the selected independent variable(s) explains the variability in the dependent variable(s).

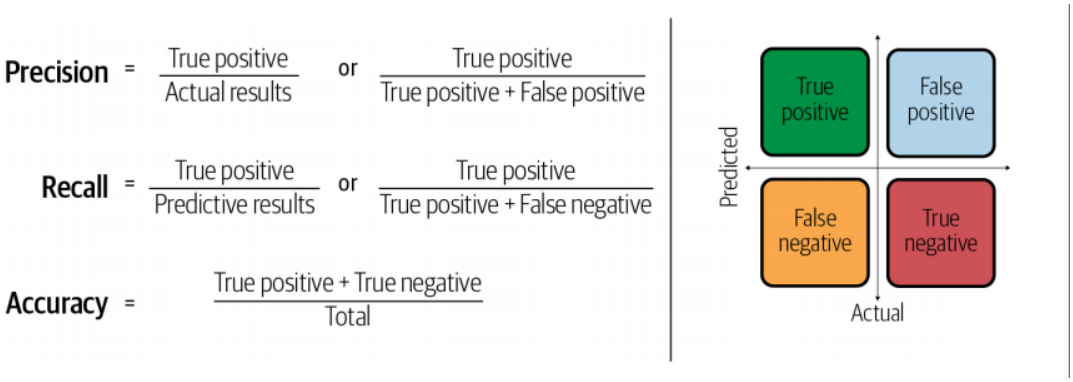
Now, let's look at the evaluation metrics for supervised **classification**.

Classification For simplicity, we will mostly discuss things in terms of a binary classification model.

some common terms are:

- True positives (TP) — Predicted positive and are actually positive.
- False positives (FP) — Predicted positive and are actually negative.
- True negatives (TN) — Predicted negative and are actually negative.
- False negatives (FN) — Predicted negative and are actually positive.

The figure below helps explain the difference between three commonly used evaluation metrics for classification problems: accuracy, precision, and recall.



Accuracy accuracy is the number of correct predictions made as a ratio of all predictions made. This is the most common evaluation metric for classification problems and is also the most misused. It is most suitable when there are an equal number of observations in each class (which is rarely the case) and when all predictions and the related prediction errors are equally important, which is often not the case.

Precision Precision is the percentage of positive data points of the total predicted positive data points. Here, the denominator is the model prediction done as positive from the whole dataset. This is a good measure when we really want to avoid false positives, such as email spam detection.

Recall Recall (or sensitivity or true positive rate) is the percentage of positive instances out of the total **actual** positive instances. Therefore, the denominator (true positive + false negative) is the actual number of positive instances present in the dataset. Recall is a good measure when there is a high cost associated with false negatives (e.g., fraud detection).

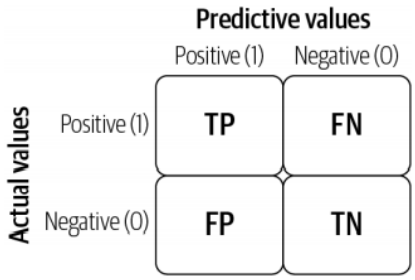
Area under ROC Curve (AUC) AUC is a metric for binary classification problems. ROC is a probability curve, and AUC is the degree of separability. It gives a measure of how capable the model is of distringuishing between classes. The higher the AUC, the better the model is at predicting zeros as zeros and ones as ones. An AUC of 0.5 would indicate no class separation capacity.

Confusion matrix The confusion matrix is a square matrix that reports the *counts* of the true positive, true negative, false postive, and false negative predictions of a classifier.

Below is a confusion matrix. The squares would contain counts for the respective categories.

The confusion matrix is a handy presentation of the accuracy of a model with two or more classes. The table presents predictions on the x-axis and accuracy outcomes on the y-axis. The cells of the table are the number of predictions made by the model. For example, a model can predict zero or one, and each prediction may actually have been a zero or a one. Predictions for zero that were actually zero appear in the cell for prediction = 0 and actual = 0, whereas predictions for zero that were actually one appear in the cell for prediction = 0 and actual = 1.

Selecting an evaluation metric for supervised classification The evaluation metric for classification depends heavily on what we want to do. **For example, recall is a good measure when there is a high cost associated with false negatives such as fraud detection**. We will further examine these evaluation metrics in several examples.



1.5 Model Selection

The following figure gives an overview of the factors that we should consider for the model selection process. As you can see, we have to make some trade-offs when we choose which model to use. There is no model that is superior in every respect.

The way we prioritze certain aspects of the model depends heavily on the problem at hand. For example, if we are using a model to reject or accept credit card applications, we need a model that is easy to intepret, we face legal action for not being able to explain how decisions were made. In this case, we would probably avoid ANNs, even though ANNs may have other features we desire.

	Linear regression	Logistic regression	SVM	CART	Gradient boosting	Random forest	Artificial neural network	KNN	LDA
Simplicity	✔	✔	✔	✔	✘	✘	✘	✔	✔
Training Time	✔	✔	✘	✔	✘	✘	✘	✔	✔
Handle non-linearity	✘	✘	✔	✔	✔	✔	✔	✔	✔
Robust to overfitting	✘	✘	✔	✘	✘	✔	✘	✔	✘
Large datasets	✘	✘	✘	✔	✔	✔	✔	✘	✔
Many features	✘	✘	✔	✔	✔	✔	✔	✘	✔
Model interpretation	✔	✔	✘	✔	✔	✔	✘	✔	✔
Feature scaling needed	✘	✘	✔	✘	✘	✘	✘	✘	✘

[ ]: