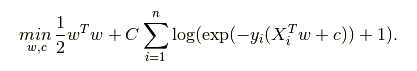
**Logistic Regression**

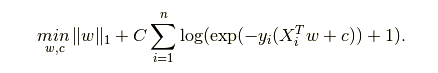
Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a [logistic function](https://en.wikipedia.org/wiki/Logistic_function).

The implementation of logistic regression in scikit-learn can be accessed from class **[LogisticRegression](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html" \l "sklearn.linear_model.LogisticRegression" \o "sklearn.linear_model.LogisticRegression)**. This implementation can fit binary, One-vs- Rest, or multinomial logistic regression with optional L2 or L1 regularization.

As an optimization problem, binary class L2 penalized logistic regression minimizes the following cost function:



Similarly, L1 regularized logistic regression solves the following optimization problem



**Random Forest**

Random Forest is a trademark term for an ensemble of decision trees. In Random Forest, we’ve collection of decision trees (so known as “Forest”). To classify a new object based on attributes, each tree gives a classification and we say the tree “votes” for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

Each tree is planted & grown as follows:

1. If the number of cases in the training set is N, then sample of N cases is taken at random but *with replacement*. This sample will be the training set for growing the tree.
2. If there are M input variables, a number m<<M is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing.
3. Each tree is grown to the largest extent possible. There is no pruning.

**XGBoost**

Another classic gradient boosting algorithm that’s known to be the decisive choice between winning and losing in some Kaggle competitions.

The XGBoost has an immensely high predictive power which makes it the best choice for accuracy in events as it possesses both linear model and the tree learning algorithm, making the algorithm almost 10x faster than existing gradient booster techniques.

The support includes various objective functions, including regression, classification and ranking.

One of the most interesting things about the XGBoost is that it is also called a regularized boosting technique. This helps to reduce overfit modelling and has a massive support for a range of languages such as Scala, Java, R, Python, Julia and C++.

Supports distributed and widespread training on many machines that encompass GCE, AWS, Azure and Yarn clusters. XGBoost can also be integrated with Spark, Flink and other cloud dataflow systems with a built in cross validation at each iteration of the boosting process.

**KNN**

It can be used for both classification and regression problems. However, it is more widely used in classification problems in the industry. K nearest neighbors is a simple algorithm that stores all available cases and classifies new cases by a majority vote of its k neighbors. The case being assigned to the class is most common amongst its K nearest neighbors measured by a distance function.

These distance functions can be Euclidean, Manhattan, Minkowski and Hamming distance. First three functions are used for continuous function and fourth one (Hamming) for categorical variables. If K = 1, then the case is simply assigned to the class of its nearest neighbor. At times, choosing K turns out to be a challenge while performing kNN modeling.

The advantages of support vector machines are:

* Effective in high dimensional spaces.
* Still effective in cases where number of dimensions is greater than the number of samples.
* Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
* Versatile: different [Kernel functions](http://scikit-learn.org/stable/modules/svm.html#svm-kernels) can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

* If the number of features is much greater than the number of samples, avoid over-fitting in choosing [Kernel functions](http://scikit-learn.org/stable/modules/svm.html#svm-kernels) and regularization term is crucial.
* SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see [Scores and probabilities](http://scikit-learn.org/stable/modules/svm.html#scores-probabilities), below).

**Naïve Bayes**

It is a classification technique based on [Bayes’ theorem](https://en.wikipedia.org/wiki/Bayes%27_theorem) with an assumption of independence between predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, a naive Bayes classifier would consider all of these properties to independently contribute to the probability that this fruit is an apple.

Naive Bayesian model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

**SVM**

It is a classification method. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a coordinate.

For example, if we only had two features like Height and Hair length of an individual, we’d first plot these two variables in two dimensional space where each point has two co-ordinates (these co-ordinates are known as **Support Vectors**)

**Decision Trees (DTs)** are a non-parametric supervised learning method used for [classification](http://scikit-learn.org/stable/modules/tree.html#tree-classification) and [regression](http://scikit-learn.org/stable/modules/tree.html#tree-regression). The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

Some advantages of decision trees are:

* Simple to understand and to interpret. Trees can be visualized.
* Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.
* The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
* Able to handle both numerical and categorical data. Other techniques are usually specialized in analyzing datasets that have only one type of variable. See [algorithms](http://scikit-learn.org/stable/modules/tree.html#tree-algorithms) for more information.
* Able to handle multi-output problems.
* Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by Boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
* Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
* Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

* Decision-tree learners can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning (not currently supported), setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
* Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
* The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
* There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
* Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

**Adaboost Classifier**

The module **[sklearn.ensemble](http://scikit-learn.org/stable/modules/classes.html" \l "module-sklearn.ensemble" \o "sklearn.ensemble)** includes the popular boosting algorithm AdaBoost, introduced in 1995 by Freund and Schapire [[FS1995]](http://scikit-learn.org/stable/modules/ensemble.html#fs1995).

The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights w_1, w_2, …, w_N to each of the training samples. Initially, those weights are all set to w_i = 1/N, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence [[HTF]](http://scikit-learn.org/stable/modules/ensemble.html#htf).

Hyper-parameters are parameters that are not directly learnt within estimators. In scikit-learn they are passed as arguments to the constructor of the estimator classes. Typical examples include C, kernel and gamma for Support Vector Classifier, alpha for Lasso, etc.

It is possible and recommended to search the hyper-parameter space for the best [cross validation](http://scikit-learn.org/stable/modules/cross_validation.html#cross-validation) score.

A search consists of:

* an estimator (regressor or classifier such as sklearn.svm.SVC());
* a parameter space;
* a method for searching or sampling candidates;
* a cross-validation scheme; and
* a [score function](http://scikit-learn.org/stable/modules/grid_search.html#gridsearch-scoring).

Some models allow for specialized, efficient parameter search strategies, [outlined below](http://scikit-learn.org/stable/modules/grid_search.html#alternative-cv). Two generic approaches to sampling search candidates are provided in scikit-learn: for given values, **[GridSearchCV](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html" \l "sklearn.model_selection.GridSearchCV" \o "sklearn.model_selection.GridSearchCV)** exhaustively considers all parameter combinations, while **[RandomizedSearchCV](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html" \l "sklearn.model_selection.RandomizedSearchCV" \o "sklearn.model_selection.RandomizedSearchCV)** can sample a given number of candidates from a parameter space with a specified distribution. After describing these tools we detail [best practice](http://scikit-learn.org/stable/modules/grid_search.html#grid-search-tips) applicable to both approaches.

Note that it is common that a small subset of those parameters can have a large impact on the predictive or computation performance of the model while others can be left to their default values. It is recommended to read the docstring of the estimator class to get a finer understanding of their expected behavior, possibly by reading the enclosed reference to the literature.

While using a grid of parameter settings is currently the most widely used method for parameter optimization, other search methods have more favourable properties. **[RandomizedSearchCV](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html" \l "sklearn.model_selection.RandomizedSearchCV" \o "sklearn.model_selection.RandomizedSearchCV)** implements a randomized search over parameters, where each setting is sampled from a distribution over possible parameter values. This has two main benefits over an exhaustive search:

* A budget can be chosen independent of the number of parameters and possible values.
* Adding parameters that do not influence the performance does not decrease efficiency.

Specifying how parameters should be sampled is done using a dictionary, very similar to specifying parameters for **[GridSearchCV](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html" \l "sklearn.model_selection.GridSearchCV" \o "sklearn.model_selection.GridSearchCV)**. Additionally, a computation budget, being the number of sampled candidates or sampling iterations, is specified using the n\_iter parameter. For each parameter, either a distribution over possible values or a list of discrete choices (which will be sampled uniformly) can be specified:

The global poverty headcount requires that the number of the poor within a country be measured by adding up the poor based on a welfare aggregate obtained through household surveys. In most countries, the aggregate of choice is per capita consumption. Indeed, 75 percent of the countries in the World Bank PovcalNet database— the official online repository of World Bank poverty data—use this aggregate. The countries in the database that use incomes are mostly high-income countries and Latin America and Caribbean countries.

Are these two aggregates—income and consumption—the same? They are not. Conceptually, income is a measure of the potential set of all goods and services that an individual or a household could obtain based on their purchasing power. Meanwhile, consumption represents a direct measure of the goods and services that the individual or household has actually obtained. Therefore, consumption does not capture opportunities, but realized outcomes that directly determine an individual or household’s well-being.

In practice, income is generally more volatile in the sense that it may be influenced greatly by seasonal factors or by a lack of regularity, particularly in agriculture and in the informal sector. It also has other important shortcomings, such as the frequent case of households that declare zero income on a survey, but exhibit a consumption level that is not zero. This may occur because the households lack income during a survey recall period, are dissaving, or are experiencing a spell of unemployment, or because the consumption of home produced goods has not been correctly measured.

In contrast, consumption does not normally vary as widely; it displays a much smoother behavior. For this reason, consumption tends to be the preferred aggregate in measuring poverty in developing economies, which typically depend more on agriculture and have a larger informal sector.

Despite these differences, both aggregates are used indistinctively in the measurement of the World Bank goals to maximize the number of countries monitored. Although this creates issues of comparability in the measurement of poverty, it allows the coverage of the global goals to be expanded. The distinction may be more problematic, however, in the analysis of inequality. This is a result of the fact that the coverage of household surveys is generally incomplete among top earners, entrepreneurial and capital incomes are inadequately reported, and measures of consumption often underestimate the living conditions of the rich.

Evaluating the performance of a model is one of the core stages in the data science process. It indicates how successful the scoring (predictions) of a dataset has been by a trained model.

Evaluating a Binary Classification Model

In a binary classification scenario, the target variable has only two possible outcomes, for example: {0, 1} or {false, true}, {negative, positive}. Assume you are given a dataset of adult employees with some demographic and employment variables, and that you are asked to predict the income level, a binary variable with the values {“<=50K”, “>50K”}. In other words, the negative class represents the employees who make less than or equal to 50K per year, and the positive class represents all other employees. As in the regression scenario, we would train a model, score some data, and evaluate the results. The main difference here is the choice of metrics Azure Machine Learning computes and outputs. To illustrate the income level prediction scenario, we will use the [Adult](http://archive.ics.uci.edu/ml/datasets/Adult)dataset to create an Azure Machine Learning experiment and evaluate the performance of a two-class logistic regression model, a commonly used binary classifier.

After running the experiment, you can click on the output port of the [Evaluate Model](https://msdn.microsoft.com/library/azure/927d65ac-3b50-4694-9903-20f6c1672089/) module and select Visualize to see the evaluation results (Figure 7). The evaluation metrics available for binary classification models are: Accuracy, Precision, Recall, F1 Score, and AUC. In addition, the module outputs a confusion matrix showing the number of true positives, false negatives, false positives, and true negatives, as well as ROC, Precision/Recall, and Lift curves.

Accuracy is simply the proportion of correctly classified instances. It is usually the first metric you look at when evaluating a classifier. However, when the test data is unbalanced (where most of the instances belong to one of the classes), or you are more interested in the performance on either one of the classes, accuracy doesn’t really capture the effectiveness of a classifier. In the income level classification scenario, assume you are testing on some data where 99% of the instances represent people who earn less than or equal to 50K per year. It is possible to achieve a 0.99 accuracy by predicting the class “<=50K” for all instances. The classifier in this case appears to be doing a good job overall, but in reality, it fails to classify any of the high-income individuals (the 1%) correctly.

For that reason, it is helpful to compute additional metrics that capture more specific aspects of the evaluation. Before going into the details of such metrics, it is important to understand the confusion matrix of a binary classification evaluation. The class labels in the training set can take on only 2 possible values, which we usually refer to as positive or negative. The positive and negative instances that a classifier predicts correctly are called true positives (TP) and true negatives (TN), respectively. Similarly, the incorrectly classified instances are called false positives (FP) and false negatives (FN). The confusion matrix is simply a table showing the number of instances that fall under each of these 4 categories. Azure Machine Learning automatically decides which of the two classes in the dataset is the positive class. If the class labels are Boolean or integers, then the ‘true’ or ‘1’ labeled instances are assigned the positive class. If the labels are strings, as in the case of the income dataset, the labels are sorted alphabetically and the first level is chosen to be the negative class while the second level is the positive class.

Going back to the income classification problem, we would want to ask several evaluation questions that help us understand the performance of the classifier used. A very natural question is: ‘Out of the individuals whom the model predicted to be earning >50K (TP+FP), how many were classified correctly (TP)?’ This question can be answered by looking at the **Precision** of the model, which is the proportion of positives that are classified correctly: TP/(TP+FP). Another common question is “Out of all the high earning employees with income >50k (TP+FN), how many did the classifier classify correctly (TP)”. This is actually the **Recall**, or the true positive rate: TP/(TP+FN) of the classifier. You might notice that there is an obvious trade-off between precision and recall. For example, given a relatively balanced dataset, a classifier that predicts mostly positive instances, would have a high recall, but a rather low precision as many of the negative instances would be misclassified resulting in a large number of false positives. To see a plot of how these two metrics vary, you can click on the ‘PRECISION/RECALL’ curve in the evaluation result output page (top left part of Figure 7).

Another related metric that is often used is the **F1 Score**, which takes both precision and recall into consideration. It is the harmonic mean of these 2 metrics and is computed as such: F1 = 2 (precision x recall) / (precision + recall). The F1 score is a good way to summarize the evaluation in a single number, but it’s always a good practice to look at both precision and recall together to better understand how a classifier behaves.

In addition, one can inspect the true positive rate vs. the false positive rate in the **Receiver Operating Characteristic (ROC)** curve and the corresponding **Area Under the Curve (AUC)** value. The closer this curve is to the upper left corner, the better the classifier’s performance is (that is maximizing the true positive rate while minimizing the false positive rate). Curves that are close to the diagonal of the plot, result from classifiers that tend to make predictions that are close to random guessing.

#### **Metrics for classification models**

The following metrics are reported when evaluating classification models. If you compare models, they are ranked by the metric you select for evaluation.

* **Accuracy** measures the goodness of a classification model as the proportion of true results to total cases.
* **Precision** is the proportion of true results over all positive results.
* **Recall** is the fraction of all correct results returned by the model.
* **F-score** is computed as the weighted average of precision and recall between 0 and 1, where the ideal F-score value is 1.
* **AUC** measures the area under the curve plotted with true positives on the y axis and false positives on the x axis. This metric is useful because it provides a single number that lets you compare models of different types.
* **Average log loss** is a single score used to express the penalty for wrong results. It is calculated as the difference between two probability distributions – the true one, and the one in the model.
* **Training log loss** is a single score that represents the advantage of the classifier over a random prediction. The log loss measures the uncertainty of your model by comparing the probabilities it outputs to the known values (ground truth) in the labels. You want to minimize log loss for the model as a whole.

### **Metrics for regression models**

The metrics returned for regression models are generally designed to estimate the amount of error. A model is considered to fit the data well if the difference between observed and predicted values is small. However, looking at the pattern of the residuals (the difference between any one predicted point and its corresponding actual value) can tell you a lot about potential bias in the model.

The following metrics are reported for evaluating regression models. When you compare models, they are ranked by the metric you select for evaluation.

* **Mean absolute error (MAE)** measures how close the predictions are to the actual outcomes; thus, a lower score is better.
* **Root mean squared error (RMSE)** creates a single value that summarizes the error in the model. By squaring the difference, the metric disregards the difference between over-prediction and under-prediction.
* **Relative absolute error (RAE)** is the relative absolute difference between expected and actual values; relative because the mean difference is divided by the arithmetic mean.
* **Relative squared error (RSE)** similarly normalizes the total squared error of the predicted values by dividing by the total squared error of the actual values.
* **Mean Zero One Error (MZOE)** indicates whether the prediction was correct or not. In other words: ZeroOneLoss(x,y) = 1when x!=y; otherwise 0.
* **Coefficient of determination**, often referred to as R2, represents the predictive power of the model as a value between 0 and 1. Zero means the model is random (explains nothing); 1 means there is a perfect fit. However, caution should be used in interpreting R2 values, as low values can be entirely normal and high values can be suspect.