**Performance Metric:**

**RMSE:**

For feature matrix , where *m* is the number of instances and *n* is the number of features in the training dataset,and prediction function (hypothesis) *h*, *Root Mean Squared Error* (RMSE) is given by the (Euclidian) norm of the prediction error :

for , (1.1)

where is the feature vector and is the label of the *i*th instance.

*Mean Absolute Error* (MAE) is given by the (Manhattan) norm:

. (1.2)

**Linear Regression:**

Linear regression model prediction is denoted as

, (2.1)

where is the parameter vector containing bias term and feature weights to , and is the feature vector corresponding to a data instance.

Training an LR model means finding values of the parameters that minimize the *cost function* of the hypothesis on a training dataset **X**, given by the MSE:

for . (2.2)

This is just the squared form of (1.1) above.

**Normal Equation:**

LR has a closed-form solution to the problem of minimizing the cost function in (2.2), called the *Normal Equation*:

, (2.3)

where is the value of that minimizes the cost function, and is the target vector containing values to .

**Batch Gradient Descent:**

Differential change in the cost function in (2.2) by changing parameter along feature axis *j* is given by the partial derivative

. (2.4)

The gradient vector contains partial derivatives along all feature axes :

. (2.5)

Gradient vector points uphill; subtracting it from gives the updated value for the next (downhill) gradient step. Step size is determined by the learning rate parameter , which multiplies the gradient vector:

. (2.6)

This *Batch Gradient Descent* formula involves calculations over the whole batch of training dataset at each gradient step, which scales poorly with the scale *m* of the training set, but well with the feature scale *n*. Training LR model with very large number of features is much faster with Gradient Descent than with the Normal Equation.

**Ridge Regression:**

Regularizing LR models (to reduce overfitting) typically means constraining the feature weights. *Ridge regression* adds a regularization term equal to half the square of norm of the weight vector , , to the cost function (2.2), forcing the learning algorithm to not only fit the data but also keep model weights as small as possible:

, (2.7)

where vector is defined as the vector of feature weights to , and is a regularization parameter: increasing reduces the weights. Note that the regularization (2.7) applies only to the feature weights (sum starts at ), and not the bias term .

As with LR, ridge regression can be performed either by computing a closed-form solution or by Gradient Descent. For the latter, just add to the MSE gradient vector (2.5). For the former, add a term to the Normal Equation (2.3), where is an identity matrix except with a 0 in the top-left cell corresponding to the bias term:

. (2.8)

**Lasso Regression:**

*Lasso Regression* adds norm of the weight vector as a regularization term to the LR cost function (2.2):

. (2.9)

One issue for Gradient Descent with Lasso Regression is that (2.9) is not differentiable at ; this can be dealt with by using a sub-gradient vector when any :

, where . (2.10)

**Elastic Net:**

*Elastic Net* is a middle ground between Ridge and Lasso Regression, where the regularization is a simple mix of both these regularization terms via a mix ratio *r*:

. (2.11)

Elastic Net become Ridge Regression for and Lasso Regression for .

**Logistic Regression:**

Just like LR, a Logistic Regression model computes a weighted sum of input features (plus a bias term), but instead of outputting the result directly like (2.1), it outputs the *logistic* of the result as a probability that an instance **x** belongs to a particular class:

, (3.1)

where is a *sigmoid function* is called logistic or logit, and outputs a number between 0 and 1.

Once the Logistic Regression model has estimated the probability , it can make the prediction if the instance **x** belongs to a positive (labeled “1”) or a negative (labeled “0”) class:

(3.2)

which follows from the fact that when .

**Cost function:**

Objective of training Logistic Regression model is to set the parameter vector so that the model estimates high probabilities for positive instances and low probabilities for negative instances . The cost function for a single instance **x** below captures this idea:

As approaches 0 for a positive class, and hence the cost function becomes very large, and it also becomes large when approaches 1 for a negative class. On the other hand, *c* is close to 0 when approaches 1 for a positive class or approaches 0 for a negative class.

Cost function over the whole training set is just the average cost over all training instances, called the *log loss*:

. (3.3)

**Gradient Descent:**

There is no close-form solution to finding the value of **θ** that minimizes this cost function (no equivalent Normal Equation). Good news is is convex, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum, if the learning rate is not too large and we wait long enough. Partial derivatives of the cost function along feature axis *j* is:

, (3.4)

and the gradient vector contains partial derivatives along all feature axes :

. (3.5)

These equations are similar to LR Gradients (2.4) and (2.5): for each instance *i*, (3.4) computes the prediction error and multiplies it by the *j*th feature value , and then averages over all training instances. Once we have the gradient vector (3.5) containing all the partial derivatives, we can use the Batch Gradient Descent algorithm similar to (2.6). For Stochastic GD we take one instance at a time, and for Mini-batch GD e use a mini-batch at a time.

**Softmax Regression:**

*Softmax Regression* generalizes Logistic Regression for multiclass classification directly, without having to train and combine multiple binary classifiers. For a given instance **x**, the idea is to first compute a *softmax score* for each class *k*:

, (3.6)

which is same as (2.1) for LR prediction; note that each class k has its own dedicated parameter vector (these vectors are stored in a parameter matrix for *K* total classes and *n* features). Next, probability that the instance x belongs to class *k* can be estimated by running the score (3.6) through a *softmax function*, which computes the exponential of the score and normalizes by dividing by the sum of all exponentials:

, (3.7)

where is a vector containing scores of all classes for the instance **x**. This is similar to the 2-class () Logistic Regression, where the probability is estimated by passing LR score through a sigmoidal function in (3.1). And just like the Logistic Regression classifier (3.2), the Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score):

; (3.8)

The *argmax* operator returns the value of the variable that maximizes a function: in (3.8) it returns the value of class *k* that maximizes the estimated probability .

Training objective for a Softmax Regression model is to estimate high probability for the target class (and consequently low probability for the other classes). The cost function for this type of classification is given by the *cross entropy*:

; (3.9)

if the target class for the *i*th instance is *k*; otherwise it is 0. This cost function reduces to log loss (3.3) for Logistic Regression when .

The gradient vector of this cost function with regards to is given by

. (3.10)

We can compute the gradient vector (3.10) for every class *k*, then use Gradient Descent (or any other optimization technique) to find the parameter matrix that minimizes (3.9):

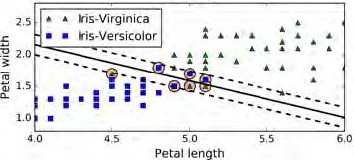
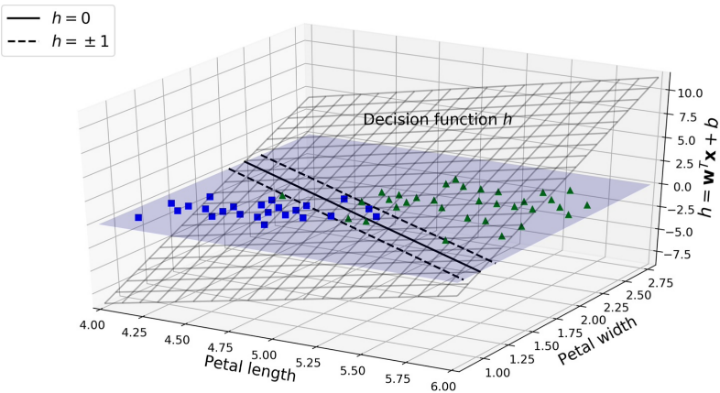
,

.

**Linear SVM Classifier:**

Linear SVM classifier model predicts the class of a new instance **x** by simply computing the decision function , where *b* is the bias term and is feature weight vector (*h* is a function in feature space); as before, is same as to (see 2.7) and *b* is . If , the predicted class is the positive class (=1), or else it is the negative class (=0):

(4.1)

Image on the left shows the 2D decision function *h* for two features of Iris dataset (this is a plane for linear SVM); image on the right is its horizontal cross-section, same as the shaded plane on left image: this plane contains the two Iris species dataset. The decision boundary (solid black line, where decision function intersects the shaded plane) is the set of points where . Dashed lines represent the points on the shaded plane corresponding to , forming a margin around the decision boundary. Training a linear SVM classifier means finding values of **w** and *b* that make this margin as wide as possible, while limiting margin violations. *Limiting* margin violation means minimizing the number of instances that fall inside the margin (“soft margin classification”), as opposed to *avoiding* margin violation which requires all instances to be outside margin (“hard margin classification”).

**Hard margin classification:**

Slope of the decision function equals the norm of the weight vector; dividing it by 2 (flattening the plane) will widen the margin by a factor of 2. So, hard margin training objective is to minimize and avoid margin violation, which requires *h* to be greater than 1 for all positive instances, and lower than for all negative instances (all instances outside margin): this means the slope of the trained decision function should be such that projections from all positive instances on the *h* plane should fall above and those from all negative instances should fall below . Defining for negative instances () and for positive instances (), hard margin classification objective for a linear SVM classifier is

, (4.2)

subject to for .

We are minimizing , or in (4.2), instead of , because has a simple derivative (equal to **w**) whereas is not differentiable at **w** = 0.

**Soft margin classification:**

To minimize (not totally avoid) margin violation requires introducing a slack variable for each instance, as a measure of how much the *i*th instance is allowed to violate the margin. We have two conflicting objectives: making ’s as small as possible to reduce margin violations, and making as small as possible to increase margin width. Hyperparameter C allows a tradeoff between these two objectives, giving us the soft margin classification objective:

, (4.3)

subject to and for .

**Quadratic Programming:**

Hard and soft margin problems are both “convex quadratic optimization problems with linear constraints”, known as *Quadratic Programming* (QP) problems. Many off-the-shelf solvers are available to solve QP problems using a variety of techniques. General formulation is given by

, (4.4)

subject to ,

where

The expression defines constraints: for , where is a vector containing *i*th row elements of **A** and is the *i*th element of **b**. Setting QP parameters as follows gives us the hard margin Linear SVM classifier objective:

* (*n* is number of features, +1 for bias term).
* (*m* is the number of training instances).
* **H** is identity matrix except 0 in top-left cell (to ignore bias term).
* , an -dimensional vector full of 0s.
* , an -dimensional vector full of 1s.
* , where is equal to with an extra bias term .

So, one way to train a hard margin linear SVM classifier is to use an off-the-shelf QP solver by passing it these parameters. Resulting parameters vector **p** will contain the bias term and the feature weights for . Similarly, QP solver can be used to train the soft margin problem.

**Kernelized SVM Classifier:**

**Dual Problem:**

Given a constrained optimization problem such as QP, also known as the *primal problem*, applying “kernel trick” (as in an SVM with nonlinear kernel) requires using a slightly different, but closely related, version called the *dual problem*. Solution to dual problem typically gives a lower bound to the solution of the primal problem, and under conditions, 1) objective function is convex, and 2) inequality constraints are continuously differentiable and convex function (both are met by SVM), dual problem can have the same solutions as the primal problem. Dual form of linear SVM objective is given as

, (5.1)

subject to for .

Once the vector that minimizes Eq.5.1 is found (using QP solver), we can compute and that minimize the primal problem by using the equation

, . (5.2)

Predictions for a new data instance can then be made with the optimized decision function:

. (5.3)

Dual problem is faster to solve than primal problem when the number of training instances is smaller than the number of features (). More importantly, it makes the kernel trick possible, while the primal does not.

**Kernelized SVM:**

Applying a 2nd-degree polynomial mapping function *ϕ* to a 2D training set transforms it into a 3D set that includes a polynomial (quadratic) feature:

. (5.4)

Applying the same mapping *ϕ* to two 2D vectors **a** and **b** and computing dot product of the resulting vectors:

.

Dot product of the transformed vectors is equal to the square of the dot product of the original vectors. It follows that if we apply the 2-degree transformation *ϕ* from (5.4) to all training instances, the dual problem (5.1) will contain the dot product , which can be replaced by ; no transformation of training instances is needed, hence no need to know the function *ϕ*, just replace the dot product in (5.1) by its square; this is “kernel trick”.

However, in going from dual to primal using Eq5.2, we still end up with equations that include in the expression for ; in fact, must have the same dimensions as , which may be huge or even infinite, so we cannot compute to make prediction with Eq.5.3. Good news is the transformed Eq.5.3 gives us an equation with only dot products between input vectors, which allows us to use kernel trick again without using (or knowing) *ϕ*:

.

Since only for the support vectors (instances that are located at the edges of the margin), making predictions involve computing the dot product of the new input vector with only the support vectors, not all the training instances. We also need to compute , using the same trick:

.

The function is called a 2nd-degree *polynomial kernel*. In Machine Learning, a *kernel* is a function capable of computing the dot product based only on the original vectors **a** and **b**, without having to compute (or even know about) the transformation *ϕ*. This follows from *Mercer’s theorem*, which states that if a function respects a few mathematical conditions called *Mercer’s conditions* (*K* must be continuous, symmetric in its arguments so , etc.), then there exists a function *ϕ* that maps **a** and **b** into another space (possibly with much higher dimensions) such that . So we can use *K* as a kernel since we know *ϕ* exists, even if we do not know what *ϕ* is.

Some of the most commonly used kernels are

.

**Decision Tree:**

*Gini* impurity of *i*th node is given as

, (6.1)

where is the ratio of class *k* instances to training instances (sample size) in the *i*th node; for a leaf node, gives the probability of the instance belonging to class *k*, and the model will predict the class corresponding to the largest value of .

Gini is default, but *entropy* impurity can be selected by setting hyperparameter criterion = “entropy”; like gini, entropy is zero for pure samples. Entropy measure is given by

. (6.2)

Gini and entropy lead to similar trees most of the time; gini is little faster, so it is a good default.

**Cost function:**

CART (*Classification and Regression Tree*) algorithm first splits the training set into two, “left” and “right”, subsets using a single feature *k* and threshold (e.g. petal length ≤ 2.45cm) that produce the purest subsets (weighted by their size). The cost function it tries to minimize is:

, (6.3)

where

Each subset is split recursively using the same logic until the maximum depth is reached, or no more split is found that reduces impurity. Unfortunately finding optimal tree is an *NP-Complete* problem that requires O(exp(*m*)) time, which is impractical even for a small training set. We must settle for a “reasonably good” solution instead.

**Computational Complexity:**

Making predictions requires traversing the tree from root to a leaf, which means going through ~ O(log2(*m*)) nodes. Since each node only requires checking the value of one feature, or probability , the overall complexity is just O(log2(*m*)), independent of the number of features. So, predictions are fast even with large training sets.

However the training algorithm compares all features (unless max\_features hyperparameter is set) on all samples at each node, which corresponds to a complexity of O(*n* × *m* log(*m*)). For small training sets (< few thousand instances) sklearn can speed training by presorting the data (set presort=true), but this slows down training for larger training sets.

**Regressor cost function:**

CART algorithm works mostly the same way as for the classifier, except that instead of trying to split the training set to minimize gini, it tries to minimize MSE. The cost function is given as

, where (6.4)

**AdaBoost:**

**Weighted error rate:**

Each training instance weight is initially set to . The first predictor is trained and its weighted error rate is computed using the general formula for the *j*th predictor:

, (7.1)

where is the *j*th predictor’s prediction for the *i*th instance. The predictor’s weight is then computed using this equation:

, (7.2)

where is the learning rate hyperparameter. The more accurate the predictor, higher is the weight. Purely random guesses result in the weight being close to 0.5, and worse (than random) prediction makes it negative. Instance weights are then updated as follows:

for . (7.3)

Then all weights are normalized (*i.e.* divided by ). The next (*j*+1th) predictor is then trained with this updated training set and so on. Algorithm stops when the desired number of predictors is reached, or a perfect predictor is found.

AdaBoost makes overall prediction for instance **x** by computing predictions of all predictors and weighing them using the predictor weights . The predicted class is the one that receives the majority of weighted votes:

, (7.4)

where *N* is the number of predictors (*k* is class index).

**PCA:**

Projecting/reducing training set matrix **X** down to *d* dimensions:

. (8.1)

Recovering (inverse transformation) training set from reduced/projected data:

. (8.2)