**Chapter 1:**

* Batch (offline) vs Online Learning
* Training on **partial set** and then using **validation set** to judge. Then train on **full training set**. Lastly you evaluate on **test set**. Also rather use **cross-validation**.
* Train on train set and then evaluate on **train-dev set**. If it shows poor performance, then it is overfitting. Otherwise now evaluate on dev set. If it shows poor performance, then it is the case of data mismatch. Lastly evaluate on test set.

**Chapter 2: Regression**

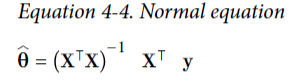
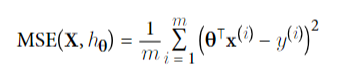
* **RMSE** gives higher weight to large errors.
* **MAE** is for data with more outliers.
* **.corr ()** in pandas measures linear correlation only.
* Shuffling data is important as some algorithms are sensitive to the order of the data they are fed.
* One issue with **OrdinalEncoder** is that the model will assume that two nearby values are more similar than two distant values. This may work for some cases but if not, use **OneHotEncoder**.
* Why not use **.get\_dummies ()** instead of OneHotEncoder? Well, the advantage of OneHotEncoder is that it remembers which categories it was trained on.
* **Scaling data** is important as models will give more importance to data with large scales and ignore the other data.
* **Normalization** is affected by outliers while **standardization** is not.
* Before standardizing data, you may need to remove heavy tails if any. You may want to apply **square root transformation** and if the tail is heavy, then instead apply **log transformation**.
* Another approach is to **bucketize** the data according to percentiles. Then replace each feature value with the index of the bucket it belongs to.
* For **multimodal distribution**, it can also be helpful to bucketize it, but this time treating the bucket IDs as categories, rather than as numerical values. This means that the bucket indices must be encoded, for example using a OneHotEncoder. Another approach is to use **RBF**.
* **RandomizedSearchCV** is often preferable, especially when the hyperparameter search space is large over **GridSearchCV**.
* Scikit-Learn also has **HalvingRandomSearchCV** and **HalvingGridSearchCV** hyperparameter search classes. Their goal is to use the computational resources more efficiently, either to train faster or to explore a larger hyperparameter space
* **scipy.stats.randint(a, b+1):** for hyperparameters with *discrete* values that range from a to b, and all values in that range seem equally likely.
* **scipy.stats.uniform(a, b):** this is very similar, but for *continuous* hyperparameters.
* **scipy.stats.geom(1 / scale):** for discrete values, when you want to sample roughly in a given scale. E.g., with scale=1000 most samples will be in this ballpark, but ~10% of all samples will be <100 and ~10% will be >2300.
* **scipy.stats.expon(scale):** this is the continuous equivalent of geom. Just set scale to the most likely value.
* **scipy.stats.loguniform(a, b):** when you have almost no idea what the optimal hyperparameter value's scale is. If you set a=0.01 and b=100, then you're just as likely to sample a value between 0.01 and 0.1 as a value between 10 and 100

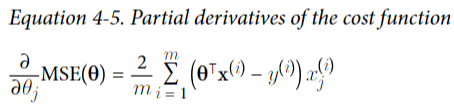
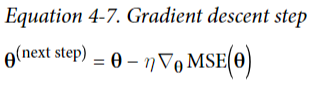
**Chapter 3: Classification**

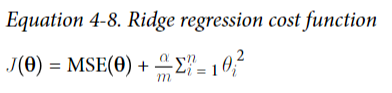
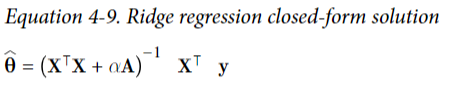
* **Stochastic Gradient Descent (SGD)** is good for large datasets and makes it well suited for online learning while **Support Vector Machine (SVM)** scale poorly on large datasets.
* **Accuracy** is not preferred metric for error in classification and especially for skewed datasets where one class is more prevalent than the other.
* Use **Precision-Recall curve** as a metric when the positive class is rare or when you are more interested in false positives rather than false negatives. Otherwise use **ROC curve.**
* **Multi-class Classification:**
* Some Scikit-Learn classifiers (e.g., LogisticRegression, RandomForestClassifier, and GaussianNB) are capable of handling multiple classes natively. Others are strictly  
  binary classifiers (e.g., SGDClassifier and SVC).
* One way is to build 1-detector, 2-detector etc. and you get the decision score from each classifier for that image, and you select the class whose classifier outputs the highest score. This is called the **one-versus-the-rest (OvR) strategy**, or sometimes **one-versus-all (OvA).**
* Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. You output the class that won the most duels. This is called the **one versus-one (OvO) strategy**. The main advantage of OvO is that it only needs data for the classes it is going to distinguish between.
* For SVC, you may want to use OvO since it can’t handle large data that graciously.
* **Multi-label Classification:**
* **KNeighborsClassifier** has the capability to do multi-label classifications.
* If you wish to use a classifier that does not natively support multilabel classification, such as SVC, one possible strategy is to train one model per label. However, this strategy may have a hard time capturing the dependencies between the labels. For this, Scikit has **ClassifierChain** which will feed appropriate labels from the previous classifiers in chain.

**Chapter 4:**

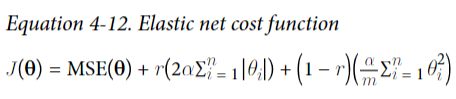
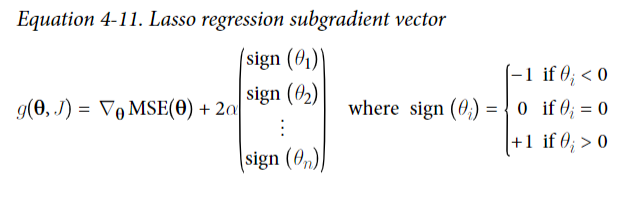
* **Linear Regression:**
* Scale the data through **StandardScaler** when using gradient descent.
* **Batch Gradient Descent** uses the complete training set while **Stochastic Gradient Descent** uses random instances of training set.
* It is wise to set a high number of epochs (training rounds) but set up an early stopping limit.
* Due to the stochastic nature of **Stochastic Gradient Descent**, the descent is very irregular, and the cost function will bounce up and down. It does not settle at a minimum. However, stochastic gradient descent has a better chance of finding the global minimum than batch gradient descent does. One solution is to gradually decrease the learning rate.
* Instead of computing the gradients based on the full training set (as in batch GD) or based on just one instance (as in stochastic GD), **mini-batch GD** computes the gradients on small random sets of instances called mini-batches. This process is less random, but it may be harder for it to escape from local minima.
* Underfitting model will not improve with increasing more data. Try to come up with a complex model. If both curves, training and validating, have reached a plateau and they are close and high, it is underfitting.
* Overfitting model will improve when you increase data. If the training curve is higher than validating curve, this means that the model is overfitting.
* **Regularization:**
* **Ridge:** 
  + Scale data before doing this regularization.
  + Specify ‘l2’ in stochastic gradient descent and make alpha = a/m where m is the no. of training examples. It is because the regularization term is (alpha / m) \* (square of the ℓ2 norm of the weight vector).
* **Lasso:** 
  + It automatically eliminates the least important features.
  + Specify ‘l1’ in sgd and keep alpha as it is.
* **Elastic Net:** It is a mixture of Lasso and Ridge regularizations.
* A little bit of regularization is always preferable. Hence, use ridge but if only few features are useful, use lasso or elastic net. Elastic net is preferred if features are greater than instances or if features are strongly correlated.

== 

 ==  

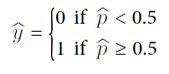
 

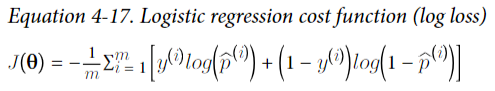
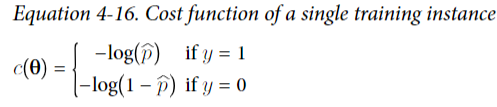
A math equation with black text

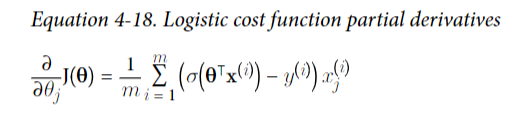
Description automatically generated with medium confidence

A close-up of a sign

Description automatically generatedA math equation with numbers and symbols

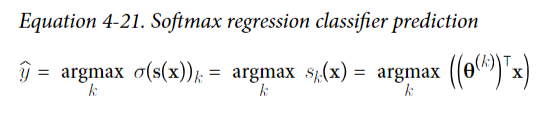
Description automatically generated

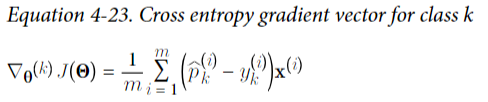
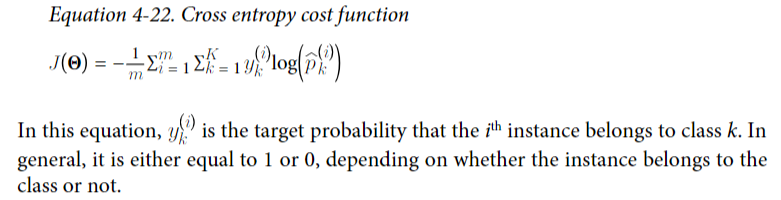


first find partial derivative of J/p and then for p/weight or bias

A white background with black text

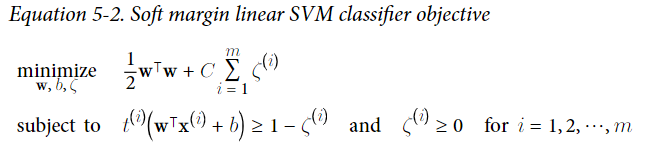
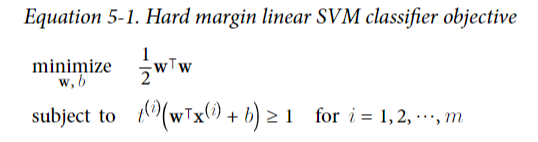
Description automatically generatedA math equation with numbers and symbols

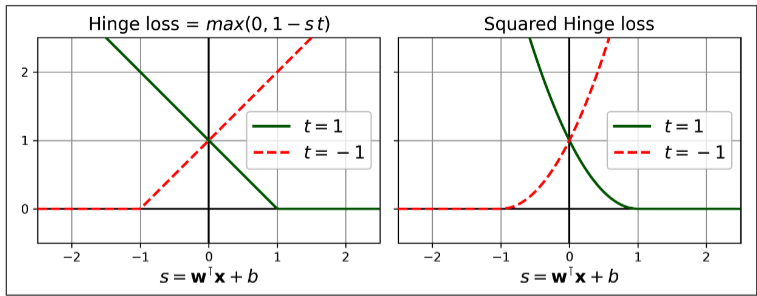
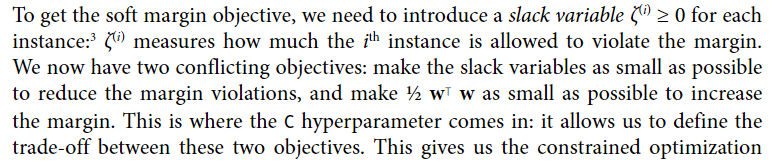
Description automatically generated

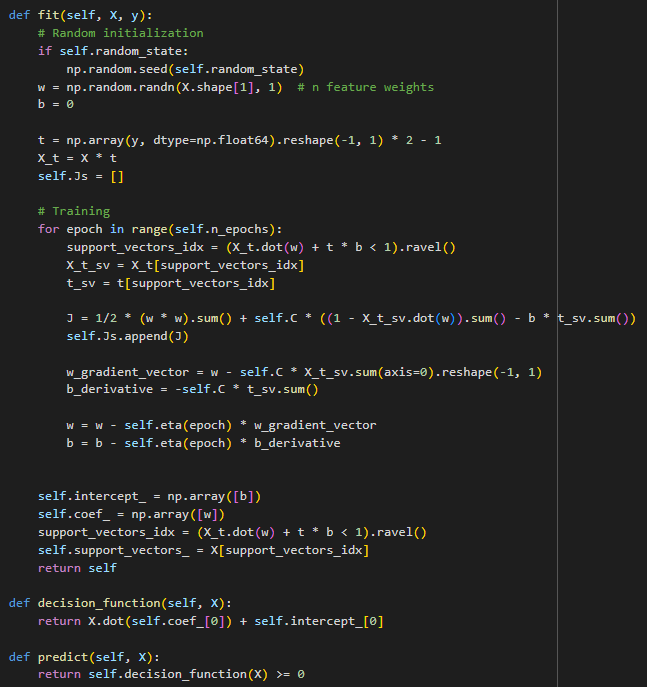


**Chapter 5: Support Vector Machines**

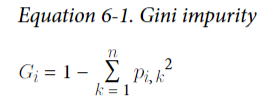
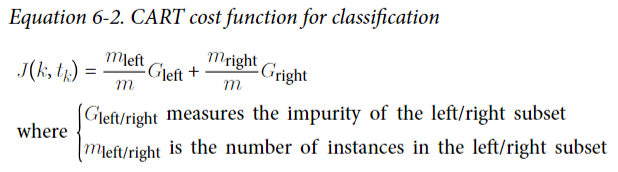
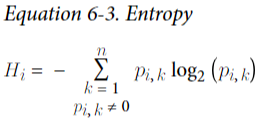
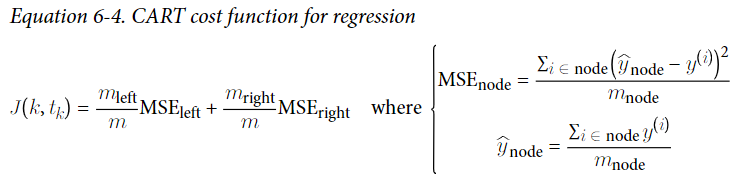
* SVM works good on small datasets. Scale the data.
* You can think of an SVM classifier as fitting the widest possible street between the two classes while limiting margin violations. This is called large margin classification. It is supported by the instances located on the edge of the street. These instances are called the support vectors.
* **LinearSVC** does not have a predict\_proba function. However, if you use **SVC**, and set probability hyperparameter to true, the function will be available.
* SVM regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instances off the street).





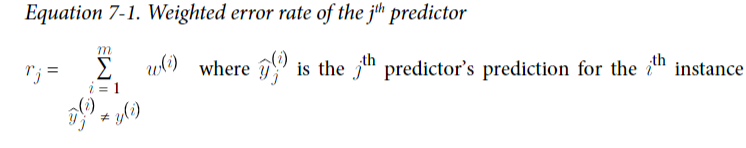


**Chapter 6: Decision Trees**

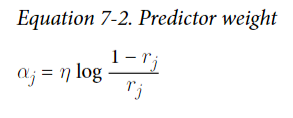
* Decision Trees do not require scaling of data.
* **pi, k is ratio of class k instances among training instances in the ith node**
* ** The algorithm works by first splitting the training set into two subsets using a single feature k and a threshold t. How does it choose k and t? It searches for the pair (k, t) that produces the purest subsets ie. by minimizing the impurity.**
* ****
* Gini impurity is faster to calculate as compared to entropy.
* **The goal for regression is to minimize MSE.**
* Decision trees love orthogonal decision boundaries (all splits are perpendicular to an  
  axis which maybe a problem. One way to limit this problem is to scale the data, then apply a principal component analysis transformation. Another main issue with decision trees is that they have quite a high variance: small changes to the hyperparameters or to the data may produce very different models. It is possible to reduce variance by averaging predictions over many trees. Such an ensemble of trees is called random forest.
* Decision Tree is a white box model which means it is explainable. On the other hand, black box models are not explainable.

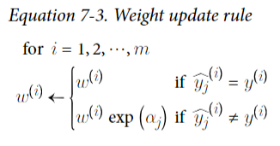
**Chapter 7: Random Forests**

* **Voting Classifiers**
  + When going with the majority voted class, you can hope for a better accuracy, but this happens if the classifiers are perfectly independent making uncorrelated errors. However, making uncorrelated errors is not possible since they are trained on the same data. One way is to train the classifiers using different algorithms.
  + Often soft voting will lead to better results than hard voting. It only works if all the estimators have predict\_proba functionality. SVC does not have this. Hence, set hyperparameter ‘probability’ of SVC to True.
* **Bagging and Pasting**
  + Bagging brings more diversity, thus, even though it has higher bias, it leads towards less variance as predictors are less correlated. Overall, bagging is preferred over pasting.
  + Bagging with sampling features brings even less variance at the cost of high bias.
* **Random Forests:**
  + It leads towards even less variance since it samples features automatically at max features = n\*\*0.5.
  + Extra-trees **may** lead to even further less variance.
* **Boosting:**
  + **AdaBoost** cannot do parallel training of predictors as it is sequentially trained. Scikit Learn uses a multiclass version called SAMME. If there are 2 classes, the version becomes AdaBoost. SAMME.R is a version that relies on probabilities rather than predictions and it generally performs better.
  + While Ada gives weight to instances, **Gradient Boosting** fits estimators on residual errors.
  + When we set subsample hyperparameter, we are doing **stochastic gradient boosting.** This method allows us to lower variance.
  + **Histogram-based gradient boosting** allows binning of input features, which in return, causes great computation speed compared to other gradient boosters.
* **Stacking:**
  + Instead of doing hard/soft voting to aggregate the predictions of predictors in ensemble, we train another model, and we pass the predictions to it.
* Normally you should use RandomForest, AdaBoost and GradientBoost initially. You may then want to try voting classifiers or stacking classifiers to push your models.

 Weights initially are set to 1/m.

The more accurate, the higher the weight.

 n is the learning rate parameter.

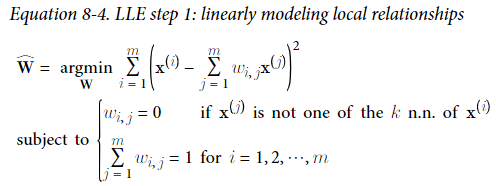
 All instances weight is then normalized i.e. weights are divided by sum of all weights

To make predictions, the predictions of all estimators are weighted using predictor weights found by Equation 7-2. Then the majority of weighted votes results in the final prediction (or the sum of weighted votes).

**Chapter 8:**

* First, you should try to train your model on original data. Then after that you can consider dimensionality reduction. It works best with many features as it may or may not give you higher performance or simpler dataset, but it will surely speed up the training. Another thing it helps us with is data visualization.
* A high dimensional dataset is more prone to be overfitted by the model.
* If you are trying to fit **PCA** yourself by using **SVD**, don’t forget to center the data by subtracting the mean from it. However, Scikit Learn’s PCA class takes care of this by itself.
* **IncrementalPCA** class of Scikit Learn is used when training set is large or when you want to apply PCA online.
* However, PCA becomes too slow when features are too many or when we have large dataset. Thus, we have another technique called ‘**Random Projection’.**
* Using random linear projection, data is projected into low-dimensional space. Surprisingly, it preserves distances between 2 instances well.
* Use **SparseRandomProjection** for large data as compared to **GaussianRandomProjection.**
* **LLE** tries to see how training instance relates to its neighbors and then it tries to unfold/unroll it maintaining this relationship.

**A math problem with black text

Description automatically generated The z is the new image in low dimension.**

**A close up of text

Description automatically generated**