Machine Learning

Shanmugha Balan S V

A Small Note

These are my notes for machine learning from a variety of sources. The main resource is Course 3 in the Applied Data Science with Python Specialization. It provided a basic route map for the rest of the document, with a large variety of other sources including the scikit-learn documentation. There are a lot of code examples to follow along, which is available in my repository at GitHub. Another source was the book "Introduction to Machine Learning with Python: A Guide for Data Scientists" by Andreas C. Müller and Sarah Guido.

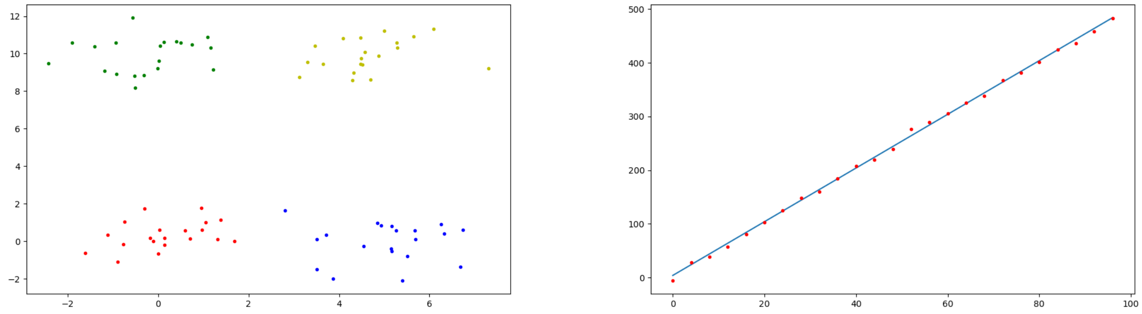
Contents

1 Machine Learning - Basics

1.1 Types of Learning

Machine Learning is broadly split into supervised learning and unsupervised learning. In supervised learning, we make the model fit to input-output pairs, so when the model is fed in with new data, it gives a similar output. The outputs are labelled in supervised learning. In unsupervised learning, the module makes arbitrary divisions, with no labelling of data. It finds clusters of similarities in the data presented to it.

Supervised learning methods mainly fall into two subcategories - classification and regression. In a classification problem, data is presented as belonging to a labelled group. It has discrete categories. Classification can be binary or multi class. Binary classification is between two groups - a positive class and a negative class. Multi class classification is when a data point should be placed in a group with many possible choices. Often, the problem is broken down into many binary classification problems. In regression, we have real number data, where we map an input to a single number output. They generally have some sort of continuity in the output or the data, and is often a matter of finding the best fit, by extrapolation or interpolation from the data.



(a) Classification

(b) Regression

Figure 1: The two types of supervised learning

1.2 Choosing a Model

A machine learning model is evaluated on the basis of a loss function, which may be constructed differently for different problems. Naturally, we would want to select the model with the best accuracy or the least loss. There are two ways we can evaluate accuracy. We can check out how well the model predicts data we have already shown it, or training data. We can also show it brand new data or testing data. To test the model, often we split the data we have into training data and testing data by shuffling it and splitting. For this, we can use `sklearn.model_selection.train_test_split`. We now train the model on the training data and evaluate it on the testing data. We evaluate many different models on their losses and accuracies. But how do we choose a split of the data we have? To do this, we can use cross validation. Cross validation is when we split the data into folds, and evaluate the model one by one keeping a fold for testing, while training on the rest. We then use the cumulative results to decide which model to pick. Crossvalidation can also be used to find the optimal hyperparameters for our model. More on the implementation of cross validation is available in the scikit-learn docs.

1.3 Generalization of an ML Model

If our model simply memorizes the training data, we may not have good results as the outliers in the training data may induce weird biases to give awkward results during testing or when the model is deployed. So our model must learn sufficiently well from the data while still retaining a good ability to generalize to new data. If our model doesn't learn well enough from the training, it is said to underfit to the data. If our model does very well in training but poorly in testing, it is said to have overfit to the data. We need to find the right spot for the model to ensure a balance between training and testing accuracies. Epochs are a training pass through the training data, where the model attempts to learn or adjust its weights and parameters. In the plot below, we can see the fitting of the model during training increases with epochs. However the testing accuracy increases to a point and then starts dropping. The phenomenon to the left of the testing maximum is called underfitting and the right side of the maximum is overfitting. For a trained model in scikit-learn, we can calculate the training accuracy as `model.score(X_train, y_train)` and the testing accuracy as `model.score(X_test, y_test)`. There are more functions for complex metrics in scikit-learn. You can also use predictive analysis with probabilities for models which support them.

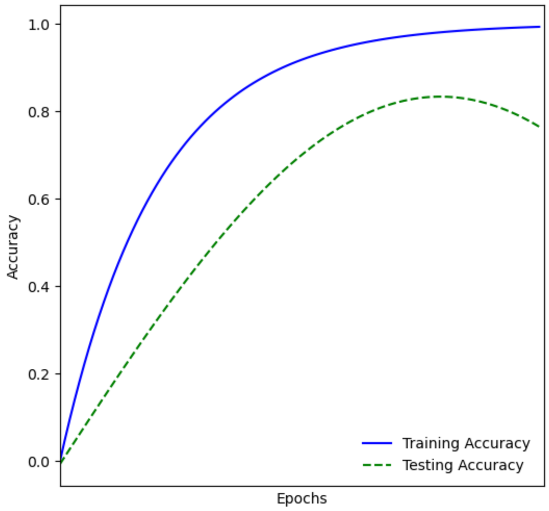
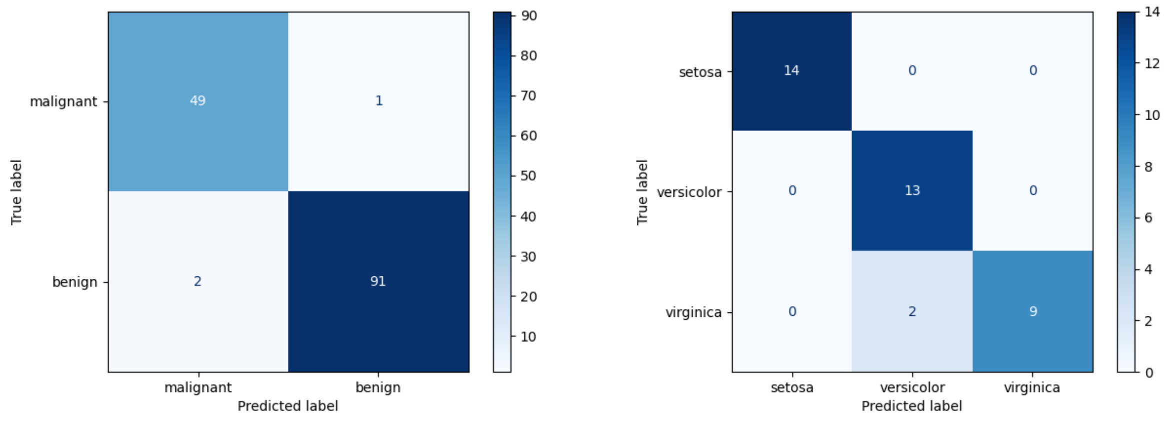


Figure 2: Training & Testing Fit for a Model

This also introduces two important terms in machine learning - bias and variance. Bias is the inability of a machine learning model to fit to the true data, in other words, it is the training error. The difference in fits between different sets is called variance. Even though the training data and the testing data come from the same underlying distribution, often they don't have the same accuracy. In relation with the previous terminology, when the model has a high bias, it is said to be underfitting the data, and when the model has a high variance, it is said to overfitting the data. We have to optimize and find the ideal model which has the least possible variance for the minimum bias.

1.4 Confusion Matrix

To analyse our model's successes and failures better, we make a confusion matrix. We can either use a normal binary two class confusion matrix, or even a multi class confusion matrix. With the example of the pre-existing datasets on the `sklearn` package, we load the iris and the breast cancer datasets. We train it with a supervised learning method - the polynomial support vector machine classifier and plot the correct classifications and the wrong classifications in a matrix. The true positives and true negatives (or correct classifications) are presented along the diagonal. Off diagonal results are the misclassfied erroneous vectors. If they are in the lower triangle of the matrix, it is a false positive, or a type I error. If they are in the upper triangle of the matrix, it is a false negative, or a type II error. Obviously, confusion matrices would be of size $n \times n$ for a n classification problem.



(a) Binary Classification

(b) Multi Classification

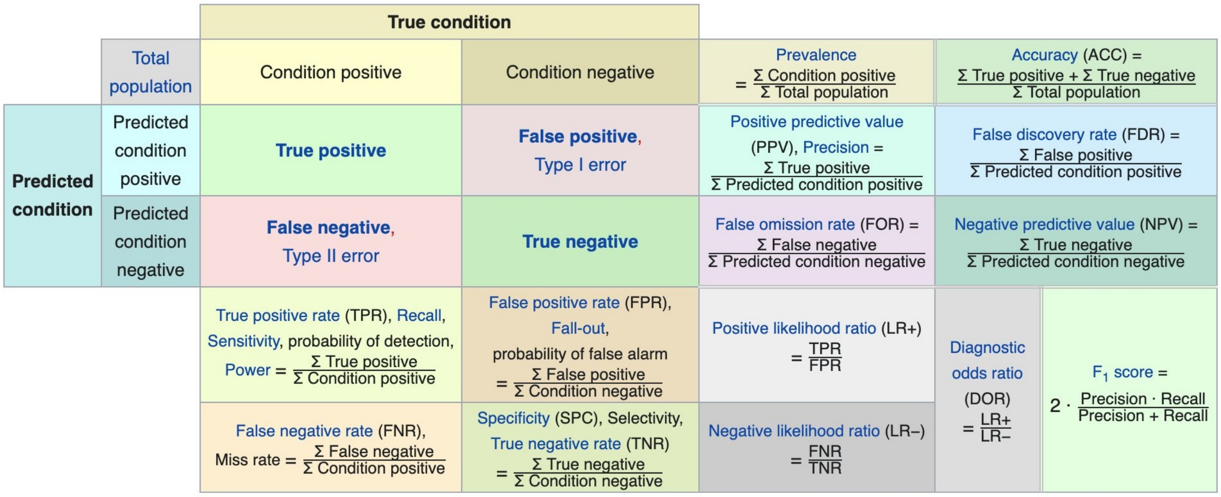
Figure 3: Confusion matrices for the breast cancer and the iris dataset

1.5 Fancier Metrics

While the overall accuracy of the model can be calculated by summing the diagonal elements and dividing it by total elements, it often doesn't tell us the full picture. We can use the "sensitivity" or true positive rate or recall. It is the proportion of a class classified correctly. Similarly, we can define the "specificity" or the true negative rate. It is the proportion of wrong examples classified correctly, and would have more meaning for a binary classification problem. Precision is the ratio of correctly classified examples to all the examples classified so. As precision and recall are the most important metrics to draw from this, we use it to give one single number the F1 score, which is the harmonic mean of precision and recall.

$$F_1 \ score = \frac{2 \times precision \times recall}{precision + recall}$$

All of these metrics are summarized in one table here.



		True condition				
	Total population	Condition positive	Condition negative	Prevalence = $\frac{\Sigma\text{ Condition positive}}{\Sigma\text{ Total population}}$	Accuracy (ACC) = $\frac{\Sigma\text{ True positive} + \Sigma\text{ True negative}}{\Sigma\text{ Total population}}$	
Predicted condition	Predicted condition positive	**True positive**	**False positive**, Type I error	Positive predictive value (PPV), Precision = $\frac{\Sigma\text{ True positive}}{\Sigma\text{ Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma\text{ False positive}}{\Sigma\text{ Predicted condition positive}}$	
	Predicted condition negative	**False negative**, Type II error	**True negative**	False omission rate (FOR) = $\frac{\Sigma\text{ False negative}}{\Sigma\text{ Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma\text{ True negative}}{\Sigma\text{ Predicted condition negative}}$	
		True positive rate (TPR), Recall, Sensitivity, probability of detection, Power = $\frac{\Sigma\text{ True positive}}{\Sigma\text{ Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\Sigma\text{ False positive}}{\Sigma\text{ Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR−}}$	F_1 score = $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
		False negative rate (FNR), Miss rate = $\frac{\Sigma\text{ False negative}}{\Sigma\text{ Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\Sigma\text{ True negative}}{\Sigma\text{ Condition negative}}$	Negative likelihood ratio (LR−) = $\frac{\text{FNR}}{\text{TNR}}$		

Figure 4: The various metrics with which a Confusion Matrix can be analyzed

1.6 ROC Curves and AUC

Often, our problem influences the parameters of a model. If we are forced to reduce the amount of false positives, or false negatives, then our model accommodates them by compensating elsewhere. Reducing false positives often increases false negatives and vice versa. We can also use this analysis to simply determine the best parameters for our model. This relationship is mapped by a plot between the true positive rate (*sensitivity*) and false positive rate ($1 - specificity$). The plot is called receiver operator characteristic or ROC graph. For imbalanced data, it would be better to use precision instead of the false positive rate. The area under the curve is another metric to help us deciding the best model to pick. The model with the largest area under the ROC curve does the best.

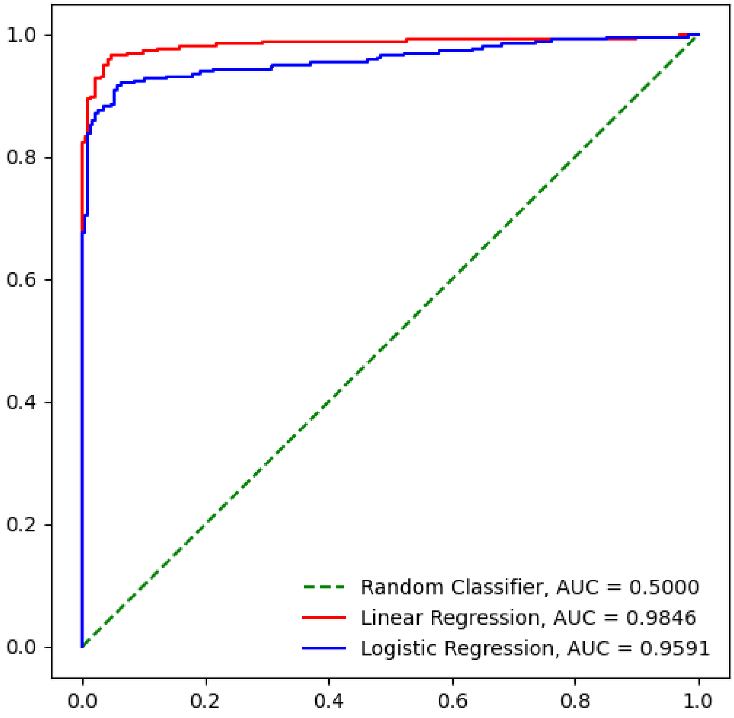


Figure 5: ROC Curve for two linear models

2 Supervised ML Algorithms

2.1 k-Nearest Neighbours

It is the simplest machine learning algorithm. It works better for classification problems, rather than regression problems. Regression kNN algorithms still exist, but are used very rarely. It memorises the training set (yes). At test time, it iterates through the training set and finds the closest match to the vector presented to it. It classifies or regresses the new vector as belonging to the same class as the closest neighbour. This model has a hyperparameter k. Instead of immediately classifying the input vector as belonging to the nearest neighbour, the model now takes a vote between the k nearest neighbours for the vector. The downside of this model is that it takes $\mathcal{O}(n)$ time during prediction, while $\mathcal{O}(1)$ time during training. Usually, this is the other way round with machine learning models, where prediction is supposed to be quick, but training can take its time. Nonetheless, it is still put to use in many character recognition or OCR tasks, where training data is quite small, and the test input is very similar to the training data.

Scikit-learn provides the class `sklearn.neighbors.KNeighborsClassifier`, which takes in an argument k for the kNN algorithm. We train this on the iris dataset from `sklearn.datasets.load_iris`. We plot it for 6 values of k, namely, 1, 3, 7, 13, 21 and 51. Note how all values of k are odd - this is to reduce the possibility of a tie during the classification. Ties can be resolved by randomly picking a class. In the figures with low k, we see islands everywhere, and there are numerous overfitting artifacts as the model is very sensitive to outliers. Larger values of k smooth things out, but in an imbalanced dataset, the majority class might swamp the classification and affect performance.

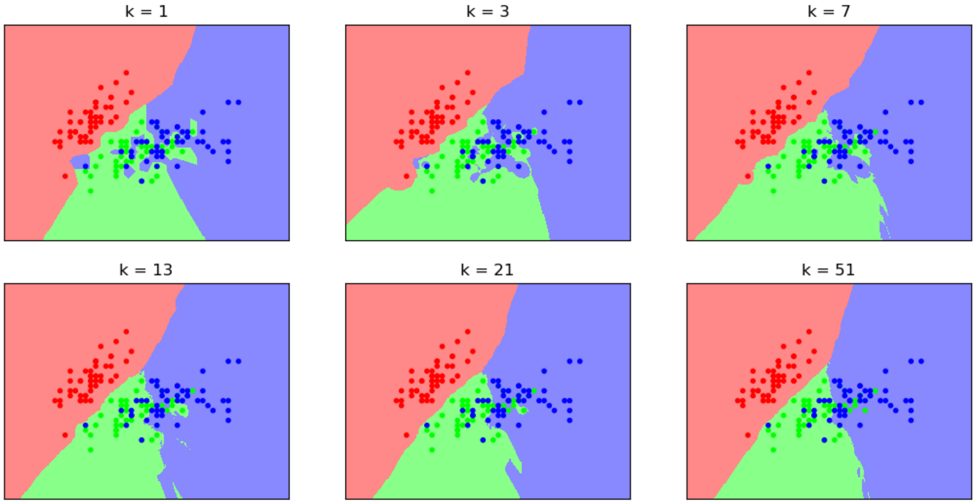


Figure 6: kNN Classification

More mathematically, the kNN algorithm takes the input vector v, and makes a pass through the training data. For every vector u_i in the training data, the norm of the difference is calculated as $l = ||u_i - v||$. The vectors u_i for which l is least is found out (by sorting or iterating or however), and the k lowest l values are pulled out separately. Among these k values, a vote of the classification data, $y_1, y_2, ..., y_n$, is taken. The majority class in these k vectors is found and the input vector is classified as $\hat{y} = y_{max}$.

2.2 Linear Models

Linear models are used widely, as they are highly versatile, and can be used for regression and classification. A linear model learns a linear function to make a prediction. So if the model has n parameters, the model predicts as below.

$$\hat{y} = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

Linear Regression

In linear regression, we try to find the line of best fit for the given data. For a set of x and y values, the seaborn library can quickly plot out the regression line with `sns.regplot(x, y)`. We also have to determine how the regression line we fit out is good. To do so, we plot out the mean line of the y data, $\bar{y}$ with x in the chart.

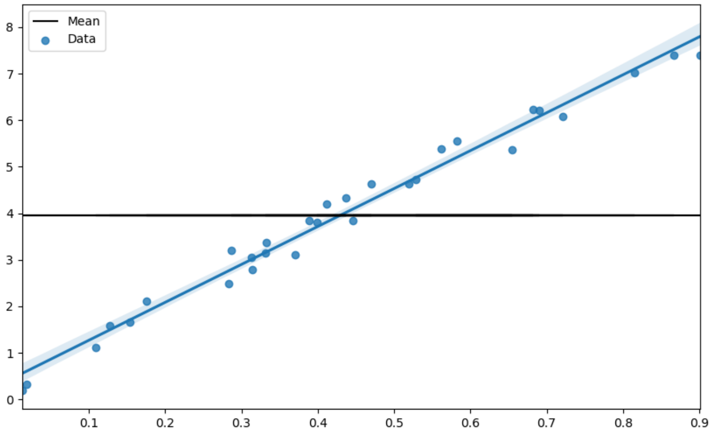


Figure 7: Linear Regression

We now draw perpendiculars from the data we have to the mean line (i.e, find the residuals) and square their distances and sum them, in other words, we find this value.

$$l = \sum_{i=1}^{n} (\bar{y} - y_i)^2$$

The value l that we found out is the sum of squared differences around the mean. Now we rotate the line to find an equation $\hat{y} = mx + c$. Now we find the new sum of squared *residuals*, with $\hat{y}$ instead of $\bar{y}$ in the above equation. The line equation with the minimum sum of squared residuals is said to be the line of best fit or the regression line.

To evaluate the goodness of fit, we have a metric called r^2. To do this, we find the sum of squared differences around the mean, l_{mean}, and the sum of squared residuals around the line, l_{line}.

$$r^2 = \frac{l_{mean} - l_{line}}{l_{mean}}$$

The r^2 score is a measure of how much of the data can be explained by the regression line. The larger the r^2 is, the better it explains the data. To explain if the r^2 is a good metric (*sigh*), we find another metric the F score. The F score is the ratio between the variation in the dependent variable explained by the line to the variation in the dependent variable not explained by the line. To find this score, we use this equation.

$$F = \frac{(l_{mean} - l_{line})/(p_{line} - p_{mean})}{(l_{line})/(n - p_{line})}$$

Here, n is the number of parameters in the data, p_{line} is the number of parameters in the regression line and p_{mean} is the number of parameters on the mean line. The various values of F form the F distribution in statistics. The F distribution can be used to find a p-value for the amount of confidence we can have in our r^2 value. As all reliable p-values must be, the F score should be small.

In scikit-learn, we can use `sklearn.linear_models.LinearRegression`. After training (fitting) a model, we can find the parameters of the model, or the equation of the line. The slope would be `model.coef_` and the y intercept is `model.intercept_`. These attributes can also be used for other linear models as well. The intercept is always a single float, but the coefficients are stored in a numpy array. The `model.score()` gives us the standard r^2 metric which we saw how to calculate above. The F score is also available in the `sklearn.feature_selection.f_regression`.

Polynomial Features

Polynomial features is a modification of regression where the existing features are mapped to a polynomial form. The problem is still a linear regression problem, but the input vector is now mapped to a higher dimensional vector which serves as a pseudo-input vector of sorts.

$$\mathbf{x} = (x_0, x_1) \rightarrow \mathbf{x'} = (x_0, x_0^2, x_1, x_1^2, x_0x_1)$$

In this example, the input vector of 2 dimensions is mapped to a 5 dimensional input. The same ordinary least squares criterion which solves linear regression can be used, but the now instead of a line, we can fit polynomial functions to the data. In the graphic below, there is the nonchalant linear regression, with some higher degree 3 and degree 7 curves. The degree 15 curve is clearly sucking up to the data and overfitting as it jiggles across the graph. The original data was a randomly scattered $sin(x)$ plot.

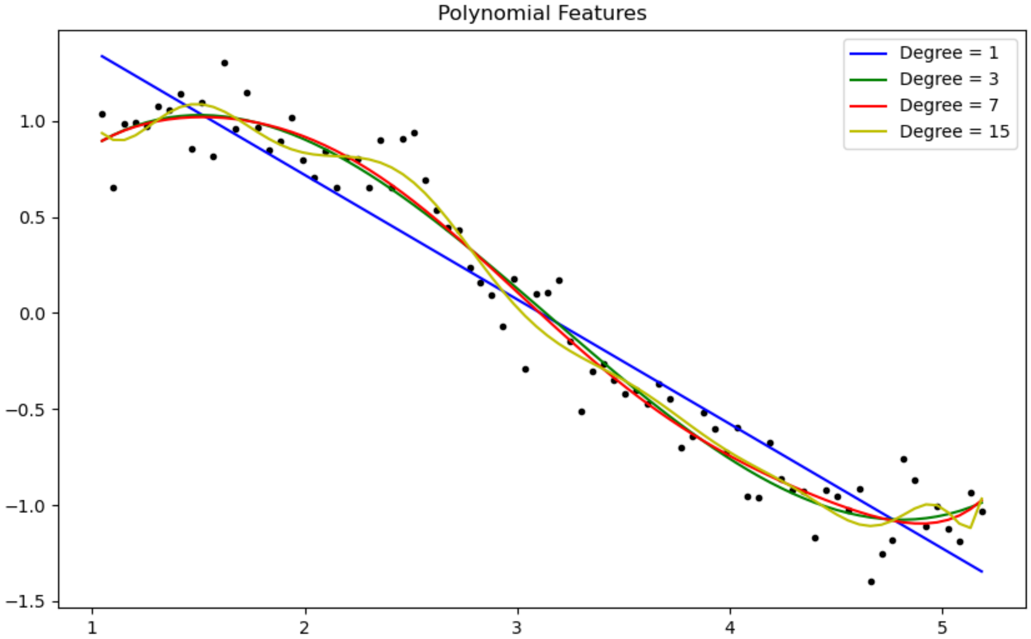


Figure 8: Polynomial Feeatures for the degrees 1, 3, 7, 15

Ridge and Lasso Regression

Ridge regression or Tikhonov regularization is a modified linear regression procedure. It adds a regularization term to the model to avoid overfitting. It reduces model complexity and forces the model to pick smaller parameters.

This reduces the variance of the errors from the mean value. Here, it uses the L_2 regularization. Adding a regularization term reduces model complexity by increasing loss arbitrarily, hence forcing the model to pick a smaller set of parameters to minimize loss. The regularization parameter of ridge regression is α which when set to 0, is just plain vanilla linear regression. Increasing α slowly imposes the regularization term on the model, and values above 1 start forcing the model parameters towards zero. The ridge regression loss equation is as below.

$$l = \sum_{i=1}^{n}(y_i - \hat{y})^2 + \alpha \sum_{j=1}^{p} w_j^2$$

The sum of squares of w_i is added to the loss function with the hyperparameter α which pushes down its values. After the training, the prediction from the model is simply plugging in X for the values of W and b.

To prevent unfair scaling of the values in cases where Y is very large compared to X or vice versa, we normalize the values. When we push the values towards zero and towards each other, features of different scales will have different contributions to the L_2 penalty. Feature normalization is done by the following equation.

$$x_i' = \frac{x_i - x_i^{MIN}}{x_i^{MAX} - x_i^{MIN}}$$

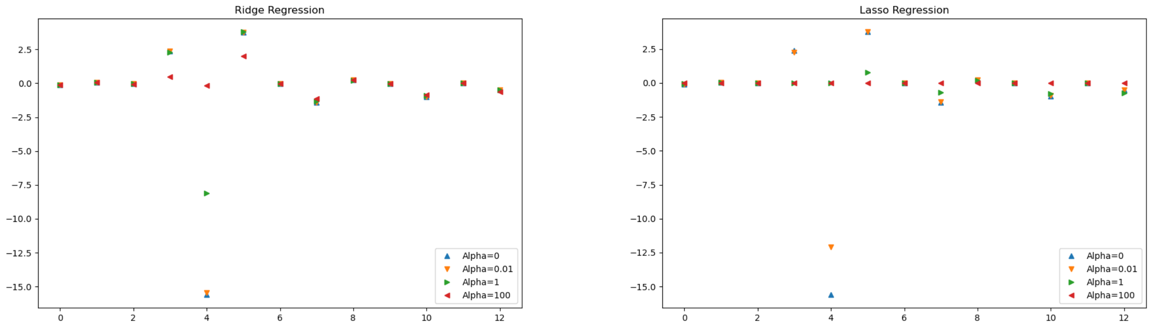
This type of normalization is called MinMax scaling, and is different from the usual standard normalization. This makes all the input features to conform to the same scale between 0 and 1. Scikit-Learn has a class for this as well, in the `sklearn.preprocessing.MinMaxScaler` which we can `model.fit_transform()`. The scaler must be fit to the training data, and should transform the test data. If the scaler is fit on the test data, it can cause data leakage and give results better than reality.

Similarly, lasso regression also has the same regularization parameter, α, only it uses L_1 regularization instead. The equation for lasso regression loss is given below.

$$l = \sum_{i=1}^{n}(y_i - \hat{y})^2 + \alpha \sum_{j=1}^{p} |w_j|$$

The parameter weights in W are set to zero for least influential variables, and this results in a sparse solution. The lasso regression model works best when there is a lot of input data, but it seems as if only very few features actually contribute to the prediction. Ridge regression is better when a lot of the features contribute little by little to the overall prediction. If we just combine all the terms (because why not) to get a giant loss function, with ordinary mean squared error, lasso regularization and ridge regression, we get the hybrid Elastic-Net Regression. These regressions have different alphas for their purposes, and setting them to 0 can result in the corresponding regressions above.

Here, we compare ridge and lasso regression, with three values of α, 0.01, 1 and 100, with vanilla regression as well. We plot out the coefficients in a graphic instead of the line to give more insight into what's happening inside. `sklearn.datasets.load_boston` was used for the data to build the models.



(a)

(b)

Figure 9: Comparison of Ridge and Lasso Regression

With some careful analysis, you can find that all the useless parameters have been set to zero. The yellow and blue markers are close to each other for the most part, and they also tend to have the most extreme values. The green marker in an awkward middle position and the red marker hugs the zero line. The effect of regularization is seen in both types of regression with the coefficients being pushed to zero for large α. Also check out scikit-learn's implementation of coordinate descent.

Logistic Regression

Unlike the other regression models, logistic regression is a classification algorithm. It is best suited to binary classification, which is done with a positive class and a negative class that is compared against a logistic curve fit to data. The logistic regression model still runs through the input features and finds an output vector, but runs the result through a non-linearity, the logistic curve. The logistic regression prediction is given by this equation.

$$\hat{y} = logistic(\hat{b} + \hat{w}_1 x_1 + \hat{w}_2 x_2 + ... + \hat{w}_n x_n)$$

$$\hat{y} = \frac{1}{1 + exp(-(\hat{b} + \hat{w}_1 x_1 + \hat{w}_2 x_2 + ... + \hat{w}_n x_n))}$$

The logistic function transforms its real valued input vector to an output between 0 and 1, we can interpret the output as the probability that the feature belongs to the positive class. we can now add a threshold and use this for classification of the data. This is similar to the sigmoid activation function in neural networks. For better visualization, let us consider two features from the iris dataset, the sepal width and sepal length. Plotting them against each other, we fit this data on a logistic regression classifier and can see the decision boundaries. We can use all features, but this multidimensional data can't be visualized as easily.

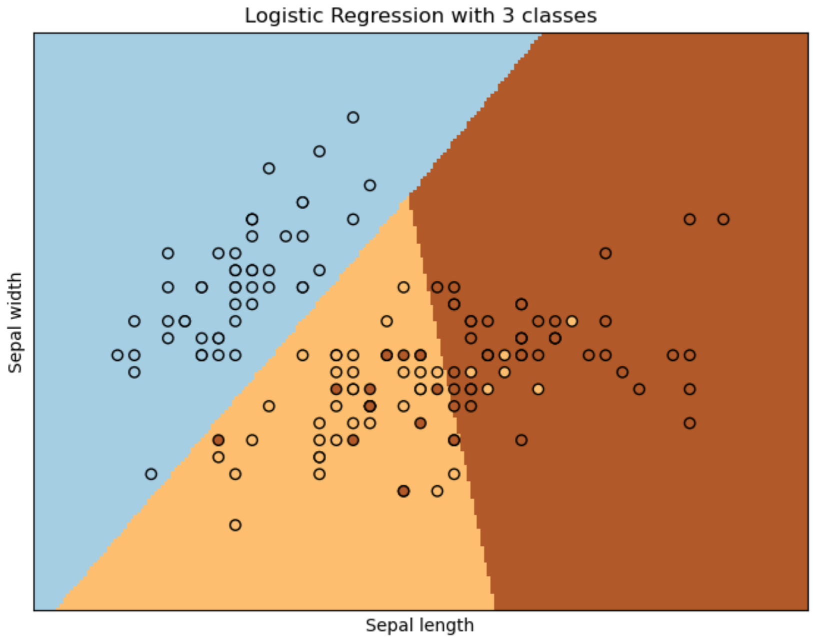


Figure 10: Logistic Regression with 2 features from 3 classes

Unlike linear regression, logistic regression doesn't use the concept of r^2 or residuals. Instead, it uses maximum likelihood estimation. A first probability curve given a set of features is made initially. The curve is used to calculate the likelihood of observing a positive class for every data point in the train set. The overall likelihood is now found by multiplying these likelihoods together. Now a new probability curve is found, and the process is repeated. The curve with the maximum likelihood is said to fit the data best, and is used for the classification. Let's try to formulate this mathematically. We first consider a logit function. W is our weights row vector, and X is the column vector of feature inputs. b as we have seen before is the bias.

$$h(X) = W \cdot X + b$$

Now we interpret the logistic function output as a conditional probability.

$$P(y = 1|X; h(X)) = \frac{1}{1 + e^{-h(X)}}$$

This can be re-written as follows.

$$WX = ln\left(\frac{P(y = 1|X; h(X)}{1 - P(y = 1|X; h(X)}\right)$$

Being a binary classification problem we're considering here, it's either a positive class or a negative class, in other words success or failure, which brings us to the Bernoulli distribution. We have a single Bernoulli trial, and is a $n = 1$ case in the binomial distribution. The number of successes is $k = 0, 1$ for a single trial, which gives us this.

$$Y \sim B(1, p) = p^k(1 - p)^{1-k}$$

Here p is the threshold probability we use in the logistic curve based prediction. Therefore, likelihood, is given by this.

$$L(w|y) = \prod_{i=1}^{n} P(Y = y_i)$$

With product being a difficult method to use, we find the log-likelihood. This gives us our loss function to **maximize** unlike most other algorithms, and we use gradient ascent to this end.

$$loss = \frac{-1}{m}\sum_{i=1}^{n} y_i log(h(X)) + (1 - y_i)log(1 - h(X))$$

`LogisticRegression` in scikit-learn gives us a parameter C for controlling its regularization, which is set to L_2 by default. Higher C implies every individual point be classified correctly, while lower C means the model adjusts to clusters over points.

2.3 Naïve Bayes Classifier

The naïve Bayes classifier is a very simple classifier that uses Bayes theorem to determine probability of a vector belonging to a class. Consider data with k classes and n features. For an input vector v of size n, we determine the probability of it belonging to a class c_i by evaluating inverse probabilities with Bayes theorem. The probability hence is as follows.

$$P(c_i|v) \propto P(c_i)P(v_1|c_i)P(v_2|c_i)...P(v_n|c_i)$$

The c_i having the largest probability is determined to be the class to which the vector v belongs.

$$\hat{y} = \arg\max_{c} P(c)\prod_{i=1}^{n} P(v_i|c)$$

This algorithm is very quick, and it allows the distribution of each feature to be analysed separately. While it is a decent classifier, it is a terrible estimator. The algorithm comes in three flavours in scikit-learn. These classes are all in the `sklearn.naive_bayes`. `GaussianNB` can be applied to any continuous data, while `BernoulliNB` assumes binary data and `MultinomialNB` assumes count data (that is, that each feature represents an integer count of something, like how often a word appears in a sentence).

The gaussian algorithm assumes a gaussian likelihood of features.

$$P(v_i|c_i) = \frac{1}{\sqrt{2\pi\sigma_{c_i}^2}} exp\left(-\frac{(v_i - \mu_{c_i})^2}{2\sigma_y^2}\right)$$

The multinomial algorithm works best in text classification, with multinomially distributed data. First a count of the number of times a feature i appears in a class c is taken.

$$N_{yi} = \sum_{x \in T} x_i$$

Then the total count of all features for the class is taken.

$$N_y = \sum_{i=1}^{n} N_{yi}$$

Finally, the relative frequency count is taken with a smoothing parameter α. It is called Laplace smoothing when $\alpha = 1$ and Lidstone smoothing when $\alpha < 1$.

$$\hat{\theta_{yi}} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

The Bernoulli algorithm is a small variant of the multinomial version, but it uses this different decision rule to penalize the non-occurrence of a feature k which is an indicator of a class c.

$$P(v_k|c_i) = P(k|c_i)x_k + (1 - P(k|c_i))(1 - x_k)$$

2.4 Support Vector Machines

Support vector machines are usually split into linear support vector classifiers (SVC) and kernelized support vector machines (SVM). It is highly effective in data with many features in large dimensions. They don't provide probability estimates directly, and is strictly a classification algorithm. As with almost every other classifier, we first take out the case of linear functions f, with the standard equation.

$$f(X) = W \cdot X + b$$

If the output of f is a positive quantity, we classify the input vector as belonging to the positive class and vice versa for the negative class. To optimize this line, we have to position the decision boundary ideally. Now we find the margin, the smallest distance between the decision boundary and the data points. We consider corresponding target values for the input vector X as $t_n \in \{-1, 1\}$. This allows $t_n f(X)$ to be positive for all data points in the training set. This margin can now be determined by finding the distance to the decision hyperplane from the data point, and that can be found by this.

$$d = \frac{t_n(W \cdot X + b)}{||W||}$$

The maximum margin solution for the SVM is now found by optimising W and b for solving this.

$$\arg\max_{w,b} \left\{ \frac{1}{||W||} \min_n [t_n(W \cdot X + b)] \right\}$$

However we have made the assumption that the data is linearly separable, which is usually not the case. For this, we have a regularization parameter C which controls how our not-so-clean data is classified. Larger values of C means less regularization, which means the model tries to fit as many individual points as possible correctly, while smaller values is more tolerant of errors, and tries to maximize margins. In other words, for the linear SVC, we can formulate the problem as minimizing l.

$$l = \frac{1}{2}||W||^2 + C \sum_{i=1} \max(0, t_n f(X))$$

Here, we can see the effect of C in the `LinearSVC` in scikit-learn which is a faster implementation of the support vector classifier for the special case of a linear kernel.

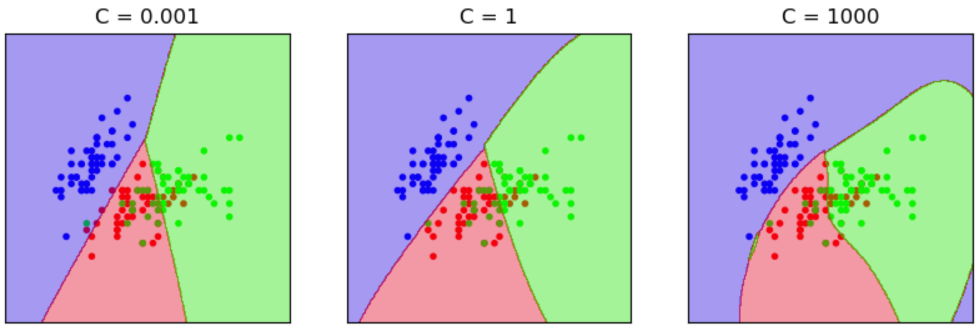


Figure 11: Effect of C on Linear SVC

Another trick to make some non linearly separable data separable is to used kernelized SVMs. We can rescale our W and b to set the distance of the points closest to the margin as 1.

$$t_n(W \cdot X + b) = 1$$

Therefore all data points must satisfy this.

$$t_n(W \cdot X + b) \geq 1$$

For each of these N points, we have constraints, which means we use Lagrange multipliers to optimize this loss. λ will be the vector with the multipliers.

$$\mathcal{L}(W, b, \lambda) = \frac{1}{2}||W||^2 - \sum_{n=1}^{N} \lambda_n \{t_n(W \cdot X + b) - 1\}$$

Setting $\frac{\partial \mathcal{L}}{\partial W} = 0$ and $\frac{\partial \mathcal{L}}{\partial b} = 0$, we obtain two conditions.

$$W = \sum_{n=1}^{N} \lambda_n t_n X$$

$$0 = \sum_{n=1}^{N} \lambda_n t_n$$

Substituting these equations back, we get this maximization with respect to λ problem.

$$\tilde{\mathcal{L}}(\lambda) = \sum_{n=1}^{N} \lambda_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \lambda_n \lambda_m t_n t_m K(x_n, x_m)$$

We have constraints, that all elements of our λ vector are positive and dotting the λ vector with t_n should result in 0. Also, in the equation, we have defined a kernel function K. This kernel function helps in making the SVM capable of learning high level features very similar to the polynomial features technique in linear regression. In the polynomial kernel, the kernel mapped is $\langle x, x' \rangle$ where x' is the higher polynomial. In the scikit-learn implementation, we can use a parameter `degree` to control the dimension of the output space. In the radial basis function kernel, the kernel is $K(x, x') = exp[-\gamma \cdot ||x - x'||^2]$. The original input feature space is transformed into an exponentially decaying function of distance between the vectors. Here a large γ means complex, tightly constrained decision boundaries, while small γ allows for a larger similarity radius. After transforming the space, it uses a linear classifier as we saw above to perform the classification. Here, let's take a look at the various support vector machine kernels with various parameters on the first two features of the iris dataset.

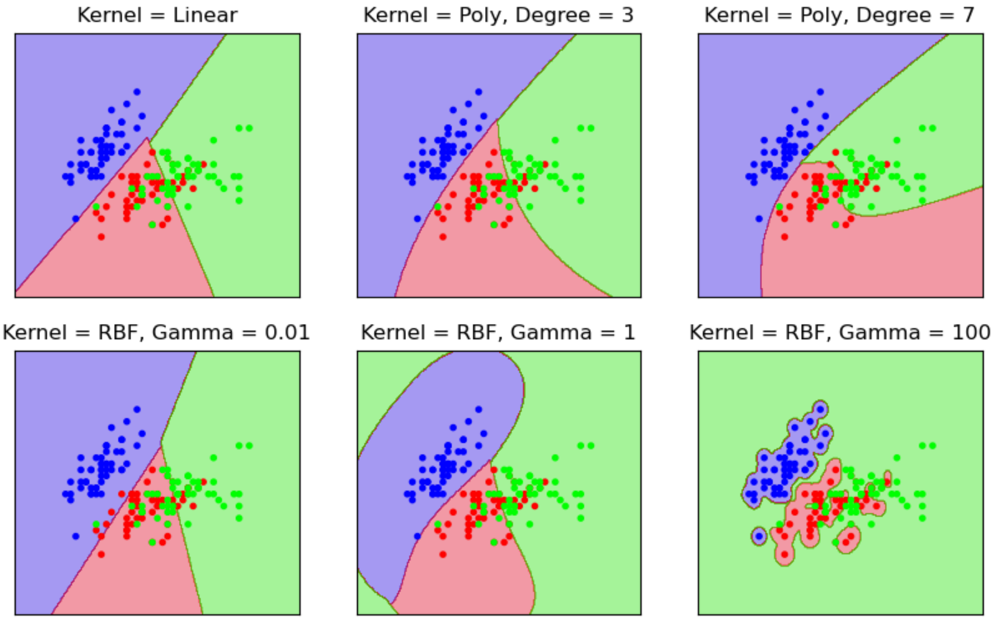


Figure 12: SVM with different kernels

Normalization of data can have a huge impact on SVM performance, so be sure to apply `MinMaxScaler()` to the data before using SVMs. While SVMs work really well for multidimensional data, their efficiency decreases as training set increases.

2.5 Decision Trees

Decision trees are powerful models which learn a series of hierarchical questions which lead to a decision. The node at the top of the tree is called root node, and the final nodes at the bottom are called leaves. The intermediate nodes and decision making is done based on the features in the training set. They are very easy to interpret and often require very little data cleaning. It can handle both numeric and categoric data. However decision trees tend to overfit and may create biased trees in unbalanced datasets. The complexity of a decision tree construction is $\mathcal{O}(n_{features}n_{samples}log(n_{samples}))$. The trees constructed need not necessarily be balanced, as they conduct a greedy search for the feature that offers the largest reduction in entropy. The query time is still $\mathcal{O}(log(n_{samples}))$.

A decision tree is constructed with three measures of impurity - entropy, misclassification or Gini index. At each decision node, we calculate the probability of the positive class and the negative class. Entropy being a measure of information, will tell us if each split is useful or not. Let's consider N classes for our data. The entropy is defined as based on the proportions of elements of each class, represented by $p_1, p_2, ..., p_N$. The entropy is given by:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

The Gini index is given by:

$$H(X) = \sum_{i=1}^{n} p_i(1 - p_i)$$

And the misclassification coefficient is given by:

$$H(X) = 1 - \max(p_i)$$

At each decision node m, let's say we have A as our data set for that node. The individual training vectors and label vectors are still represented by X and y. We have the split as $\theta = (i, t)$ with i the feature being considered for the split and t the split threshold. We split the data as:

$$A_{left} = (X, y) | x_i <= t$$

$$A_{right} = A - A_{left}$$

Now we calculate the node impurity with the impurity functions defined above.

$$G(A, \theta) = p_{i,left} H(A_{left}) + p_{i,right} H(A_{right})$$

If pick the parameters which minimise impurity.

$$\theta * = \arg\min_{\theta} G(A, \theta)$$

Continue this recursively until the maximum depth is reached, or the new split doesn't look better than the previous one.

Classification is just a simple stride down the tree, which can be interpreted very easily. We can even visualize the tree on scikit-learn with `sklearn.tree.plot_tree(model)`. To train the model, we can again use the same module but we call the classifer, `sklearn.tree.DecisionTreeClassifier()`.

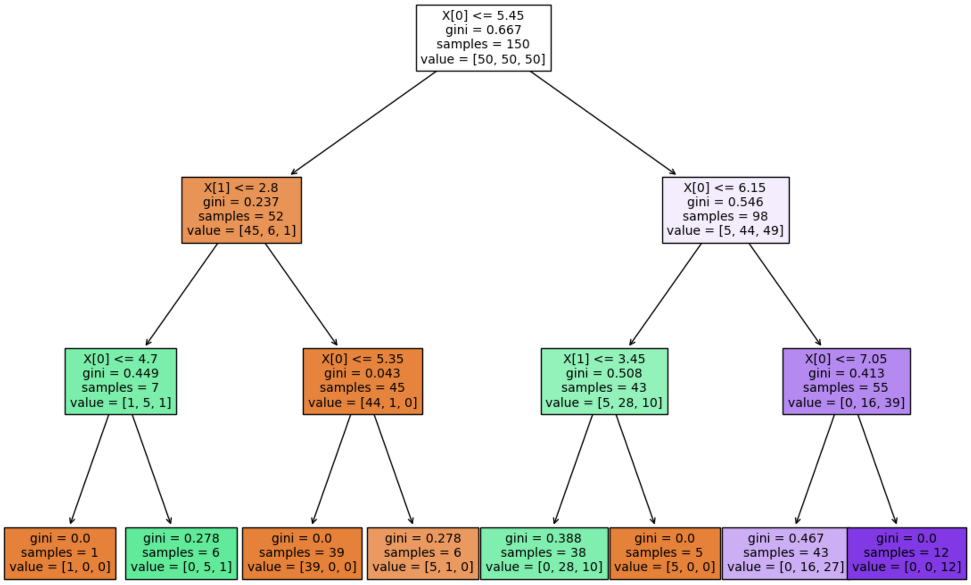


Figure 13: Depth 3 Decision Tree on the iris dataset

Decision trees can also be used for regression problems. For regression problems with a decision tree in scikit-learn, use `sklearn.tree.DecisionTreeRegressor()`. At a node m with N_m observations, we can use the mean squared error or L_2 loss to minimize mean and to determine splits. X is the training data at the node in question.

$$\bar{y}_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$

$$H(X) = \frac{1}{N_m} \sum_{i \in N_m} (y_i - \bar{y}_m)^2$$

We can also use the mean absolute error or L_1 loss, but now we use the median instead of the mean.

$$\tilde{y}_m = \underset{i \in N_m}{median}(y_i)$$

$$H(X) = \frac{1}{N_m} \sum_{i \in N_m} |y_i - \bar{y}_m|$$

One drawback of decision trees is that they love to overfit to the training data. Some dimensionality reduction technique would help combat this. Converting sparse data to a better format would also improve training time. Another common technique is to limit the depth of the tree. This is called pre-pruning. Here, we can look at the tree overfitting heavily in the default settings. The plot of the default tree is here. The decision surfaces for decision trees look rectangular because of the sharp thresholds in the decision nodes.

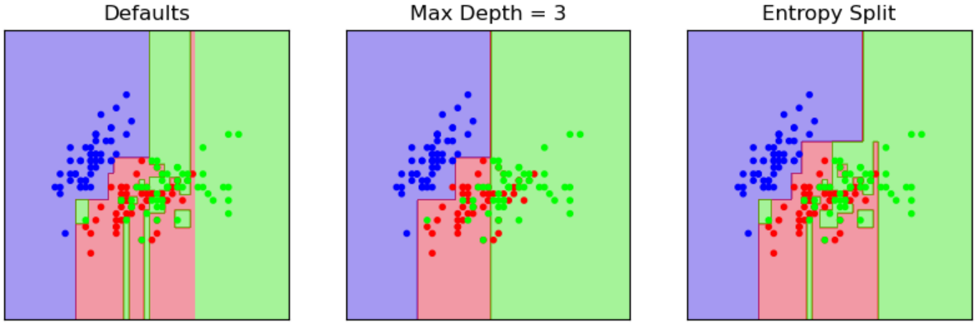


Figure 14: Decision Surfaces for Decision Trees

Minimal cost-complexity pruning is another algorithm used to cut out leaves in the tree to prevent overfitting. For every possible decision tree with pruned leaves, we evaluate the cost complexity measure. Here, T is the tree, n_L is the number of leaves in the tree and $J(T)$ is an method to calculate misclassification, in simpler terms a loss function. α is a tunable hyper parameter, which must be found through cross validation.

$$J_\alpha(T) = J(T) + \alpha n_L$$

To find the optimal α, first we start with a tree built on all the data, and start with $\alpha = 0$. Now we increase α till pruning the leaves gives us a lower J_α. We now test the different trees with cross validation at their optimal αs and find which tree does well consistently across the cross validation.

Random Forests

Random Forests are another method to counter overfitting in decision trees. We build multiple trees, and use their collective decision to decide the classes. First, we make a bootstrapped dataset, by bootstrap aggregation or bagging. First, we draw a bootstrap sample $Z*$ of size N from the training data $Z = (X, y)$. Now we grow a decision tree T_i with $Z*$. We repeat this process n times if we want to populate our forest with n trees. If we wanted to make a classification, then we take the majority vote across all the n trees in the forest. If we want to make a regression prediction, we do so by:

$$\hat{y} = \frac{1}{n} \sum_{i=1}^{n} T_i(X)$$

In scikit-learn, the random forest implementation is `sklearn.ensemble.RandomForestClassifier()`. Scikit-learn also has a regressor in `sklearn.ensemble.RandomForestRegressor()` or a collection of totally random trees in `sklearn.ensemble.RandomTreesEmbedding()`.

AdaBoost

AdaBoost is one of the boosted tree learning approaches which also combines the learning from many weak learners to produce an accurate result. AdaBoost uses stumps, which are decision trees, but with only one decision node, which is also the root node. AdaBoost, which is short of Adaptive Boosting, does the adaptation with a sample sample weight column to the data. This will tell us which stumps did a better job, and all also help us make better classification stumps. Initially we set the weighting of each sample equally, as $1/N$ where we have N samples in the data.

$$w_i = 1/N$$

Now we use the Gini impurity, or entropy to calculate the best split of data. Let's say this classifier stump we trained is $T(x)$. We find the total error in the best split. It can be found by summing up the weights of all the incorrectly classified samples and normalizing it.

$$e = \frac{\sum_{i=1}^{N} w_i I(y_i \neq T(x_i)))}{\sum_{i=1}^{N} w_i}$$

This will be 0 for a perfect stump, and 1 for the worst stump. The total error is also used to weight stumps which are better at classification with a higher say in the final vote. The amount of say a stump has is given by this measure, where e is the total error:

$$a = \frac{1}{2} \log \frac{1-e}{e}$$

Finally, we update the weights of the samples, by increasing weights for the incorrectly classified samples. Sometimes, the other weights are reduced to maintain a constant weight for the sum total of weights, but that is unnecessary if it will be normalized.

$$w_i' = w_i \exp\left(aI(y_i \neq T(x_i)\right) \; \forall \; i \in (X, y)$$

We now repeat this process for as many stumps as we want to have in our forest.

AdaBoost is a meta-learning algorithm, because it uses information about incorrectly classified samples to build better copies of the classifier. In scikit-learn, AdaBoost is implemented for classification problems with `sklearn.ensemble.AdaBoostClassifier()` and `sklearn.ensemble.AdaBoostRegressor()` for regression problems.

Gradient Boosting

Gradient Boosting is similar to AdaBoost, but we construct trees of fixed sizes instead of stumps. With the example of a binary classification problem, first we calculate the log-odds of a class. We now convert this to a probability with a logistic function.

$$p_+ = \frac{e^{\log{(+/-)}}}{1 + e^{\log{(+/-)}}}$$

This gives us the predicted probability for the positive class. Now we calculate the residuals with this probability for every sample in the dataset. Now we build a decision tree of fixed size, by limiting the total number of leaves in the tree. Each leaf will have a set of the residuals from the dataset. To make a transformation of these into probabilities, we first calculate leaf output value with this formula:

$$o = \frac{\sum r_i}{\sum p_i(1-p_i)}$$

p_i is the previous probability for every sample in the leaf which is initially just the log-odds we initialised the model with. r_i is the residual for every sample in the leaf. Finally, we calculate the probability by:

$$p = logistic(p_+ + lr \times o(x_i))$$

We use a learning rate to control our algorithm and prevent it from overfitting. We repeat the above process by calculating new residuals, then growing a new tree, and continue till we reach a user defined limit or no progress is being made.

Gradient Boosting is implemented in scikit-learn as `sklearn.ensemble.GradientBoostingClassifier()` for classification and `sklearn.ensemble.GradientBoostingRegressor` for regression.

Here there is a decision boundary plot of random forests, AdaBoost and gradient boosting trained on two features of the iris dataset.

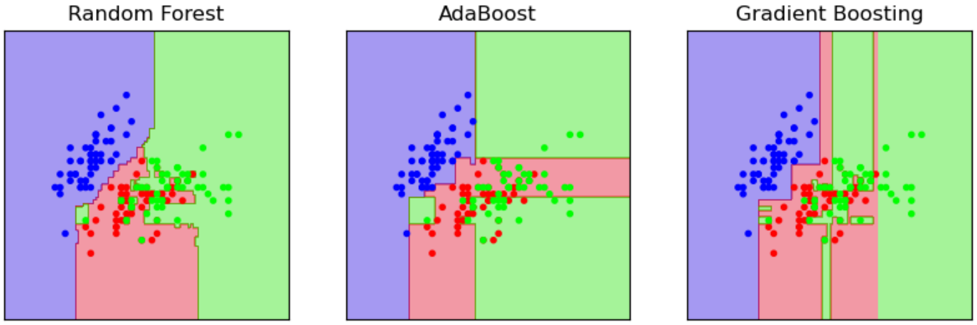


Figure 15: Comparison of Random and Boosted Decision Trees

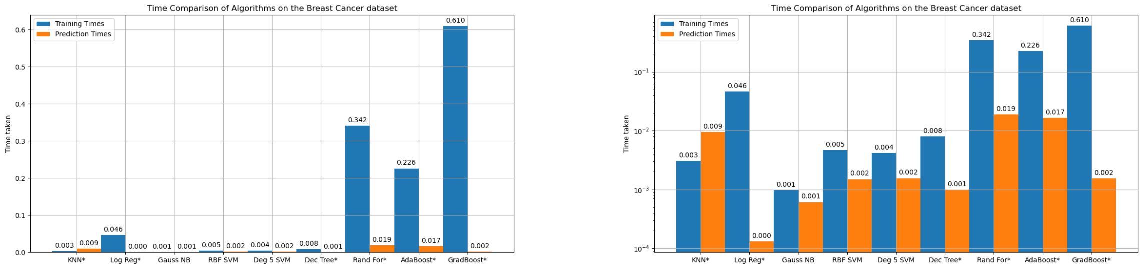
2.6 Neural Networks

We have one of the most powerful techniques for learning, and in scikit-learn, `sklearn.neural_network` has a set of functions for this. Scikit-learn supports multi-layer perceptron feed forward networks. But this whole concept deserves its own separate set of notes, so do go check those out here.

2.7 Summary

- **Nearest neighbors** are good for small datasets, good as a baseline, easy to explain.
- **Linear models** are a go-to as a first algorithm to try, good for very large datasets, good for very high dimensional data.
- **Naïve Bayes** are meant only for classification. It is faster than linear models, good for very large datasets and high-dimensional data. Often less accurate than linear models.
- **Support vector machines** are more powerful for medium-sized datasets of features with similar meaning. Require scaling of data, sensitive to parameters.
- **Decision trees** are very fast, don't need scaling of the data, can be visualized and easily explained.
- **Random forests** almost always perform better than a single decision tree, very robust and powerful. Don't need scaling of data. Not good for very high-dimensional sparse data.
- **Gradient boosted decision trees** are often slightly more accurate than random forests. Slower to train but faster to predict than random forests, and smaller in memory. Need more parameter tuning than random forests.
- **Neural networks** can build very complex models, particularly for large datasets. Sensitive to scaling of the data and to the choice of parameters. Large models need a long time to train.

Here's a comparison of time taken by some different models, in a linear scale and a log scale. The breast cancer dataset from scikit-learn was used. For plots of the wine dataset or the iris dataset, check out this link.



(a) Linear Y axis

(b) Logarithmic Y axis

Figure 16: Comparison of Supervised Learning Classifiers

And the score comparison for the same dataset:

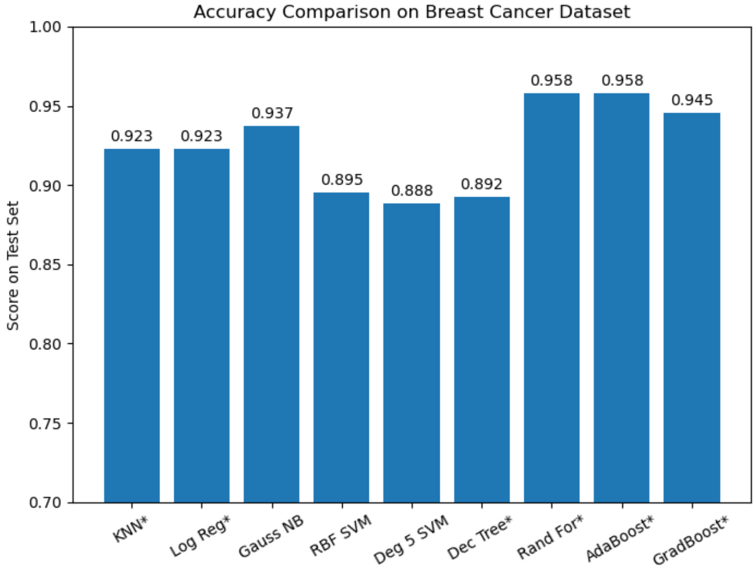
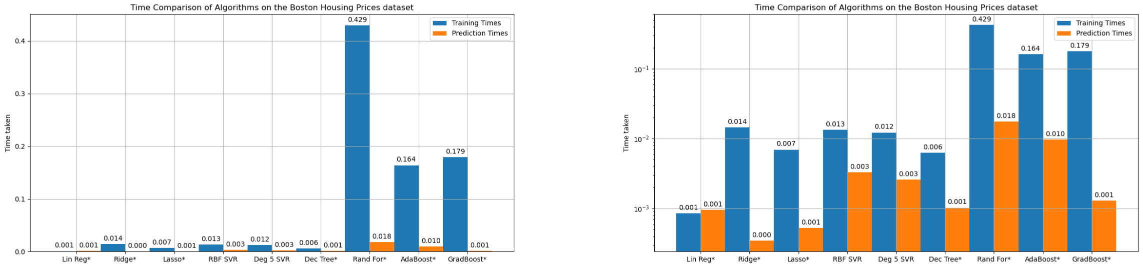


Figure 17: Score comparison for Supervised Learning Classifiers

Similarly, here's a comparison for regression. The Boston housing Prices dataset was used.



(a) Linear Y axis

(b) Logarithmic Y axis

Figure 18: Comparison of Supervised Learning Regressors

SVRs perform abysmally, with the lack of scaling for the input features.

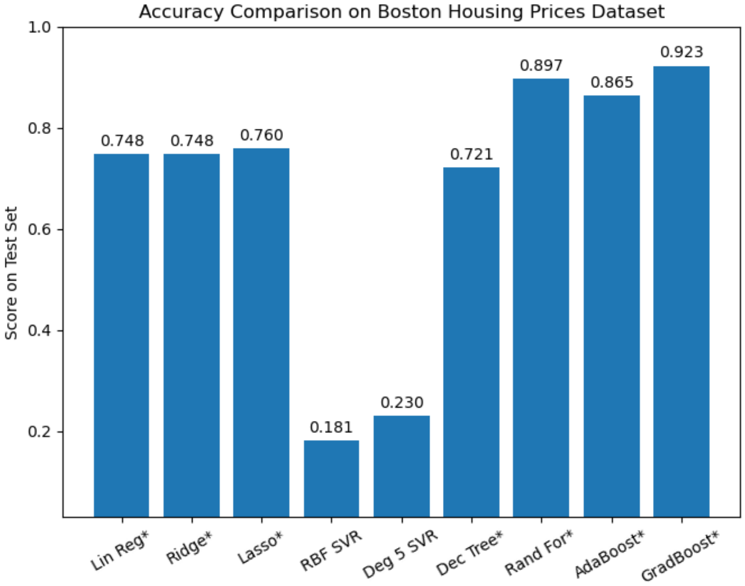


Figure 19: Score comparison for Supervised Learning Regressors