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.

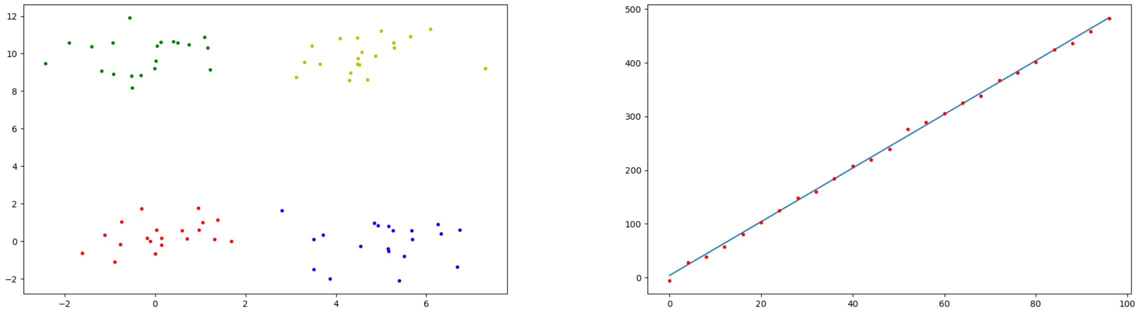
Contents

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.

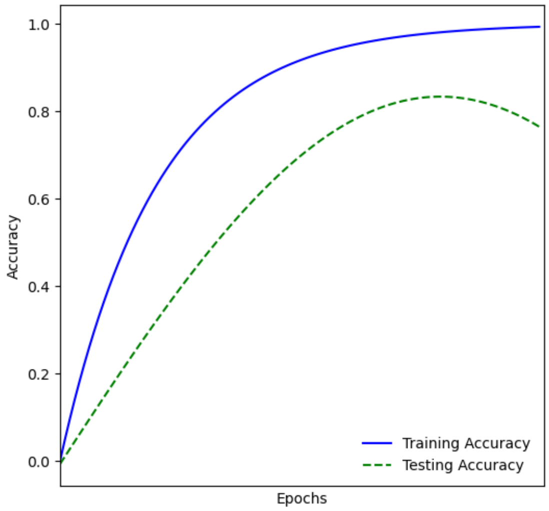
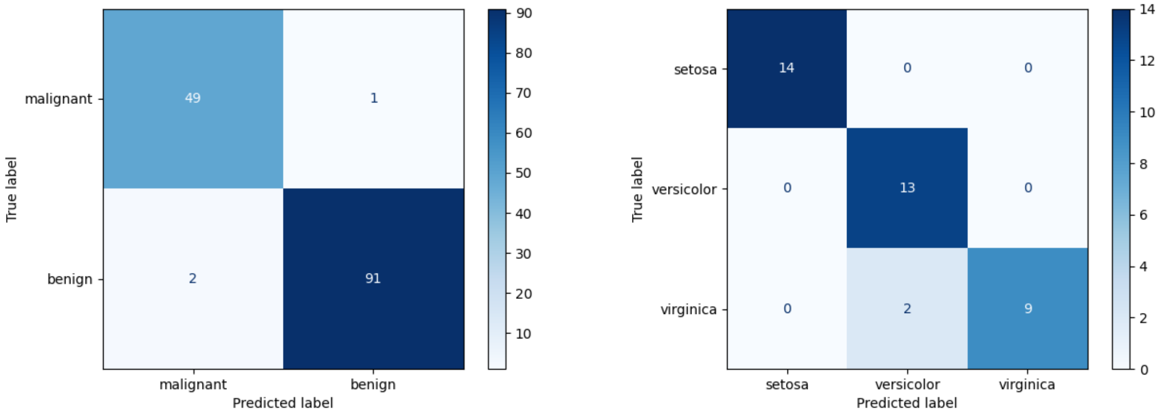


Figure 2: Training & Testing Fit for a Model

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

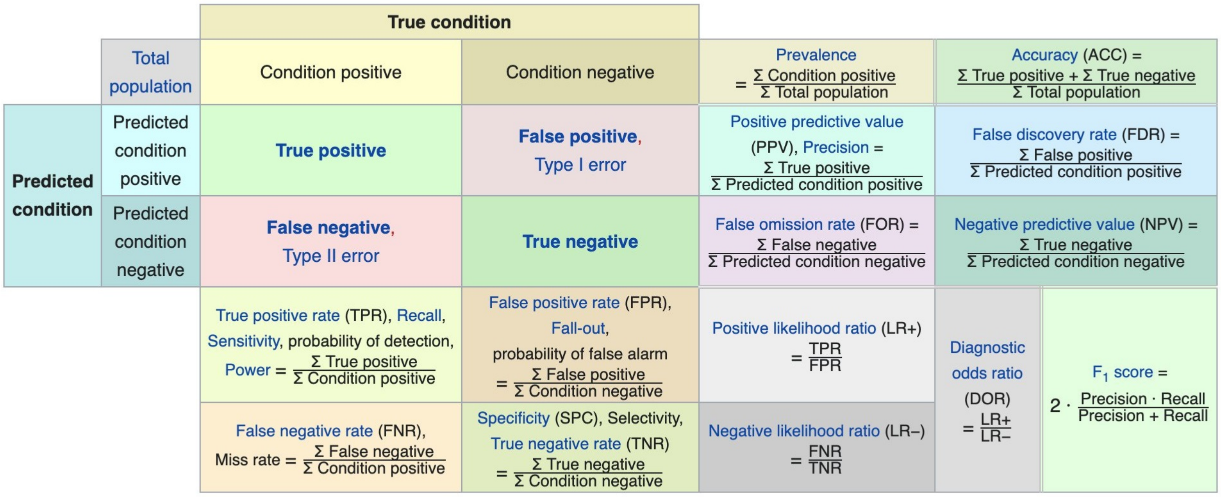
Figure 3: Confusion matrices for the breast cancer and the iris dataset

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 = $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision + Recall}}$
		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.

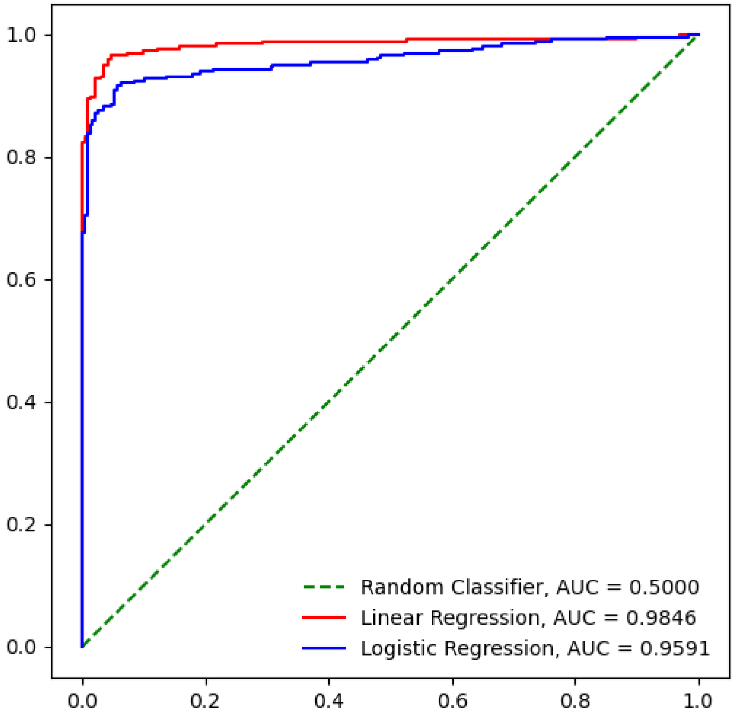


Figure 5: ROC Curve for two linear models

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.

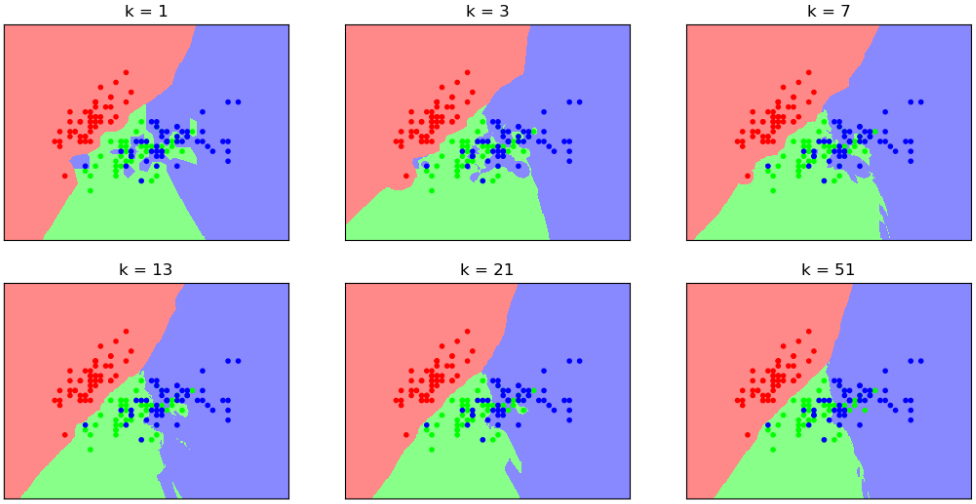


Figure 6: kNN Classification

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.

$$\hat{y} = w_1 x_1 + w_2 x_2 + ... + w_n x_n + b$$

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.

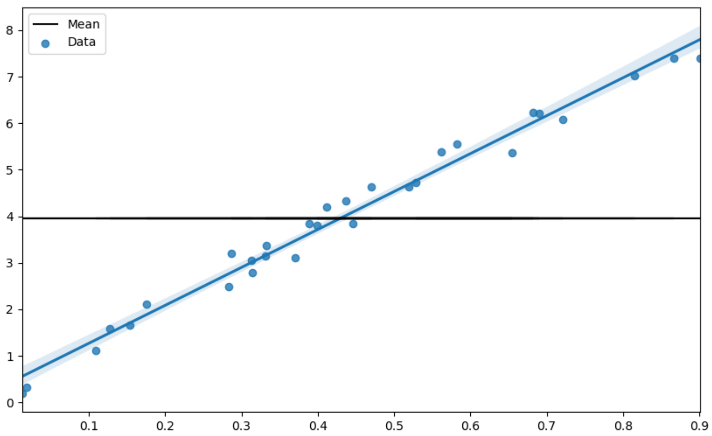


Figure 7: Linear Regression

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.

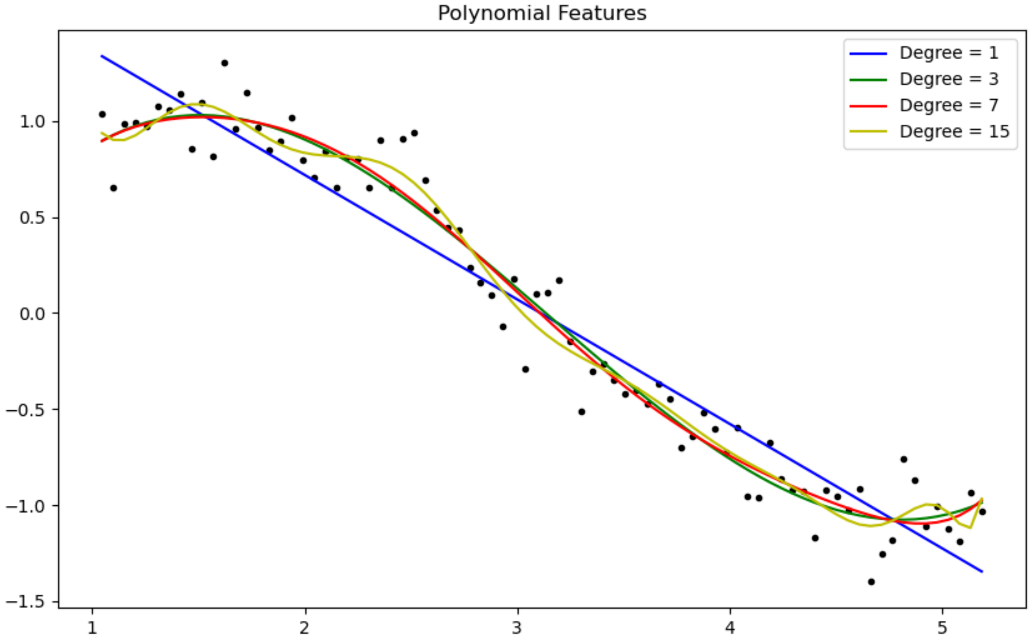


Figure 8: Polynomial Feeatures for the degrees 1, 3, 7, 15

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.

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

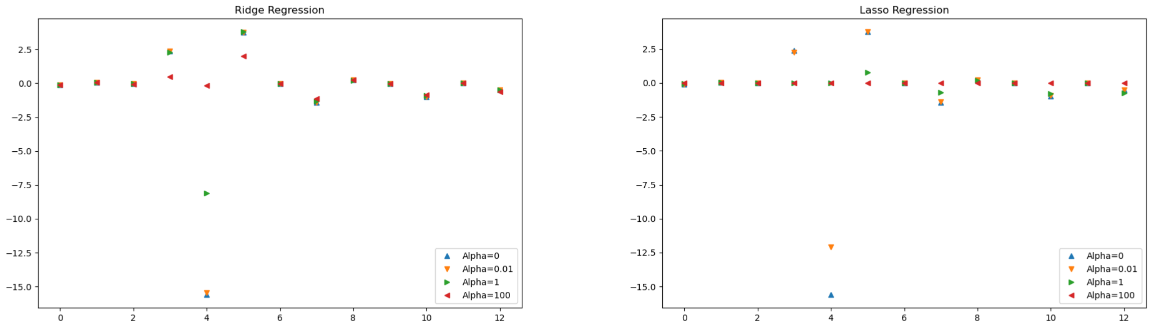
This type of normalization is called MinMax scaling, and is different from the usual standard normalization. This makes all the input features to conform to the same scale between 0 and 1. Scikit-Learn has a class for this as well, in the `sklearn.preprocessing.MinMaxScaler` which we can `model.fit_transform()`. 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.

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.

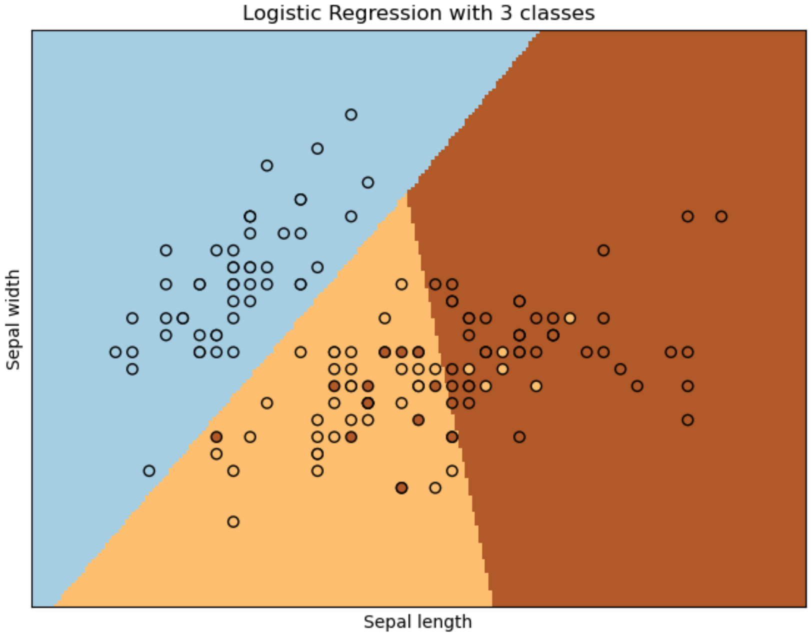


Figure 10: Logistic Regression with 2 features from 3 classes

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.

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

Logistic Regression 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.

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

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.

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