Abstract

The data set used for all benchmark applications included in this report is from a Taiwan banking institution who'd like to predict a client's likelihood of defauling on their credit card.

Multivariate Linear Regression

`Linear Regression` is a statistical learning algorithm used within a supervised setting which can be used for predicting a continuous numerical value. Linear regression can be split into two different types: `univariate` or simple regression and `multivariate` regression. What differentiates these two variations of the algorithm are whether it's fit to multiple independent variables for prediction. `Univariate` models use only one independent variable as a predictor, while `multivariate` models use multiple to estimate a prediction. This algorithm predates its modern machine learning application significantly, as it is considered one of statsitics most fundamental mechanisms.

In a nutshell, linear regression fits a line or hyperplane to the data. This regression line represents the best fit for measuring a constant rate of change between a dependent variable or `response variable` and one or multiple independent variables or `features`. The slope of the regression line is dependent on coefficients or weights which are independently assigned to the features within the model. For example, if our objective was to predict the temperature on a given day, one feature which may intuitevly seem like a candidate for a high weight is the time of the year (season). As we can assume that the season most likely has a high correlation with the response variable or in other words is an important factor in deriving its value mathematically. It is important to consider that for linear regression to produce accurate predictions, some assumptions must be met. Here are the two most important assumptions that should be met:

- The data should follow a linear pattern.
- The features should not be too correlated with eachother (`multicollinearity`).

Equation

The equation of simple regression is certainly one most people have encountered in their grade school math curiculum, as it is simply the equation for slope, or change of a dependent variable as a ratio of change in an independent variable. The equation is as follows:

$${y = \beta_0 + \beta_1x}$$

y represents the response variable, *x* the independent variable, and beta zero and beta one indicate the intercept coefficient and the coefficient for the single feature respectively.

The equation for multivariate linear regression is just an extension to this equation with a beta coefficient for every feature included.

Linear Regression's Objective Function

In order to fit a line that best matches the pattern of the data, linear regression, just as all other learning algorithms do, depends on an objective or cost function which acts as a feedback loop to the algorithm

indicating how good of a job it is doing and by how much. By doing this, the algorithm can iteratively adjust its coefficients to minimize the output of the objective function and subsequently improve its fit. One common objective function used is `RSS` or residual sum of squares. Which penalizes predictions with large residuals by squaring their value. A residual is the difference between a prediction value outputted by the model and an actual value which we figuratively keep out of the model's site until it's time to evaluate its performance. The RSS is than just a sum of the residuals for all predictions. The equation looks as follows:

Pros and Cons of Linear Regression

Here are some points for what has made this algorithm stay in the mix for so long as well as some drawbacks that might make it not the best tool for a particular regression task.

The Pros:

- Its interpratability and easy-to-grasp mathematical form make it a popular go to for baseline modeling and pedagogy.
- Performs very well on data which does follow a linear pattern.
- There are extensions and variations of linear regression, such as interaction and polynomial terms and ridge and lasso regression which can extend the models flexibility and mitigate some of its other drawbacks.
- Compared to many other models, linear regression is very computationally efficient due to its straightforward algorithm.

The Cons:

- One major drawback to linear regression is that it assumes independence of the features. This is the same issue as multicollinearity. This means applying linear regression requires vary careful feature selection.
- The relationship between a response variable and its feature is somewhat rarely a linear relationship meaning that added complexity might need to be added in the form of polynomial terms, etc.
- Linear regression is very sensitive to outliers which can severly skew predictions.

Applying Linear Regression in Python

In order to create a benchmark demonstration of applying linear regression using Python, I worked with both the `scikit learn` and `statsmodels` modules. While typically this data set is used for classification, in this application I will predict the value of `LIMIT_BAL` which represents the amount of total credit provided to a client at the household level, using a selection of the other features. I selected my variables based on two important criteria: the `Pvalue` of a feature, as well as whether or not a feature had a high level of correlation with other features (`multicollinearity`).

Nextly I fit the data to the statsmodels `OLS` class in order to view a summary output of the coefficients and their Pvalue's. OLS stands for ordinary least squares is an implimentation of linear regression which uses RSS to optimize the fit of the regression line. The summary indicated the following:

Thanks to the detailed `summary()` method from statsmodels OLS class we have validated the statistical significance of the included features and can be confident that these features are related to the response variable. Fitting and predicting using scikit learn's linear regression class on a train/test split of the model results in an RSS of `67958479153053.086`, see the code for this finale prediction below:

```
X_train, X_test, y_train, y_test =
train_test_split(X,y,test_size=0.3,random_state=123)

lr = LinearRegression()

lr.fit(X_train,y_train)

y_preds = lr.predict(X_test)
```

Logistic Regression

`Logistic regression` is also a classical statistical learning model which offers an easily interpretable algorithm, but instead for classification tasks. The most common application of logistic regression is binary classfication where the objective is to predict the probability of a certain event occuring (target variable) given a set of features. The event of interest can be represented either by a discrete binary label or a probability ranging from 0 to 1. Often the probability value as discretized by using a probability threshold (typically 0.5) where any probability equal to or greater is consider a positive response label (typically 1).

Equation

In order to make predictions ranging from 0 and 1 to model probability of the event of interest occuring, we need a function which outputs such values. The aforementioned function is called logistic regression and is the function of logisitic regression. It's formula is as follows:

Logistic Regression's Objective Function:

include explanation about what is linear about logistic regression as well

The beta coefficients must be estimated so that logistic regression's output predicts probabilities of the likelihood of an observation to fall in the positive response class which are as close as possible to the observed labels (0 or 1). In the case of this data set, the event of interest is credit card default in the following month (`default.payment.next.month`). The mathematical function which can be maximized to determine these optimal coefficient values is called `maximum likelihood` formulated as follows:

Pros and Cons of Logistic Regression:

The Pros:

- Similar to linear regerssion, logistic regression is a relatively easy model to implement and interpret.
- It can be extended to handle multiple-class classification using multinomial regression or other variants of logistic regression.
-

The Cons:

-

Applying Logistic Regression in Python

Decision Tree Classifier:

A `decision tree` is a popular algorithmic method for dividing a feature space into discrete partitions or regions through a series of conditional branches (decisions) which ultimately allow us to derive a prediction for both regression and classification use cases. Tree-based models are extremely popular in machine learning both for their effectiveness and their interpretability. Parallels are often drawn that tree-based models imitate the process of human decision making. Here we'll only consider a tree classifier as it is more relevant to the credit card default dataset. The only large contrast between regression and classification trees is in how predictions are determined from a subregion of the feature space. In classification, the predicted label is that of the majority class within the segmented region of the feature space.

To understand more closely how this algorithm works, a quick description of a decision trees structure is important. A tree can be thought of is a series of consecutive logical splits. These splits are termed branches or internal nodes. Each split consists of a logical expression with discrete binary response possibilities. For example, within the context of the credit card default dataset one such branch could consist a conditional statement like so `LIMIT_BAL >= 1000`. For all observations for which this statement is `True`, they would continue down left branch. For those which don't meet this criteria (`False`), they go down the right branch. A node in the tree that is at a dead end, so as to say it has no subsequent branches is considered a terminal or leaf node. These nodes are what allow us to classify an observation. If it falls into a leaf node in which the postive class (1) is the most prevalent than the algorithm will classify it as belonging to the positive class. Just like all machine learning models the decision tree is motivated using a cost function. The criteria for the number of splits in a tree is contingent or whether or not another split would result in better predictions (an improved objective function for each leaf node). Another important distinction to make is the order in which branches occur. The algorithm scans through the feature space and determines what splits best divide the data into the most homoegenous regions possible. Complete homogenity would mean only one class appearing in that region which would mean it is possible to make an accurate prediction. The term for this homogenity within leaf nodes is node purity.

Often times decision trees tend to become overly fit to the specific complexities of a training set. When this happens many branches will be created. This is often a symtpom of an overfit model. One way to counter act this with a decision tree is by either setting a parameter to limit the maximum number of branches allowed, or by systematically removing branches in the order in which removals cause the least increase in error.

Decision Tree Classifier's Objective Function:

Three different measures can be used as a decision tree classifier's objective function. The first of which is `classification error rate`. This measure is very easy to compute and is defined as follows:

Here error is determined by subtracting the proportion of observations of the majority class to all other observation within a terminal node from one. Intuitively the higher the proportion of observations of the majority class the smaller the error. However, the drawback to using classification error rate becomes clear when we have a large decision tree with many branches. Many branches results in leaf nodes with smaller number of observations. Imagine there're only two observations within a leaf node, in this case the classification error rate can only ever be 0 or 50.

The two alternatives to classification error rate are `the Gini index` and `binary cross entropy` or log loss for short. The Gini index works similar to classification error rate but instead multiples the porpotion of each class by one minus this porportion and then sums this across all different classes. The Gini index comes from econometrics and is usually used to represent wealth inequality within countries where a value of one equals completely inequality and zero perfect equality. In this context a Gini index of one indicates an extremely unpure node and a Gini index of zero indicates a completely pure node. The equation for the Gini index is as follows:

Log loss is the inverse logit function and can be used to optimize logistic regression. It takes on a value of zero when nodes are completely pure and one when they're completely impure just like the Gini index. It's defined like so:

Pros and Cons of Decision Tree Classifiers:

The Pros:

- Highly interpretable and easy to conceptualize.
- They're a great baseline model as we can deduce things like feature importance by applying them.
- Very minimal data preprocessing required (depending on software libraries).
- Non-parametric.

The Cons:

- Decision trees are highly prone to overfitting training data causing large variances in test error.
- More computationally intensive than the previous models.
- Decision trees have several hyperparameters and often require resource intensive tuning.

Applying Decision Tree Classifiers in Python

Random Forest Classifier

Random forests are an example of an `ensemble` learning method meaning they use a collection of learning algorithms in making predictions. The random forest as its name suggests is an ensemble of decision trees. The algorithm instantiates a large number (commonly between 100-500) different decision trees and then uses all of their individual predicted classes to tally a vote. The class with the most votes becomes the model's prediction. The idea here is that while decision trees are highly sensitive to the specific patterns within a trainset, if we instead use many trees with randomly sampled training observations as well as shuffled predictors to ensure different feature importances than we can mitigate this unwanted overfitting.

The brilliant aspect about how a decision tree works is that it invokes a sort of diversification of results in order to account for the variance of the invidivdual trees (estimators). This concept is what truly revolutionized tree-based approaches, by mitigating the overfitting drawbacks associated with a decision tree. However, the added complexity of a random forest as compared to a decision does come at a cost. The primary issue is computation time, considering each individual estimator is a decision tree in itself. Additionally, due to the increased number of moving pieces and required considerations, random forest has some important hyperparameter that need to be tuned properly.

Random Forest Classifier's Objective Function:

The objective function for a random forest classifier is the same as that for a decision tree classifier and must be optimized at the node level. The two objective functions or criterions are the Gini index and entropy as previously explained in the decision tree classifier section.

Pros and Cons of Random Forest Classifiers:

The Pros:

- Robust to variances in training data (avoids overfitting).
- Just like the decision tree classifier, random forests allow you to see feature importances explicitly.
- Very minimal data preprocessing required (depending on software libraries).

The Cons:

- Very computationally intensive and can run very slow with many estimators.
- Random forests lose a bit of the interpretability of simple decision trees in their large-scale complexity.
- Lots of thought must be given to optimizing the hyperparameters.

Applying Random Forest Classifiers in Python

Support Vector Machines:

In simplest terms an `SVM` is a statistical learning model which fits a hyperplane within feature space to classify data points discretely. While other versions of the SVM model exist for regression uses cases, such as the support vector regressor, I'll focus solely on the classical usage of the SVM model. To explain the core mechanism behind SVM, I'll focus on a simple example. Imagine you have a dataset that consists of two discrete classes. One class could be circles and the other could be squares. Now, imagine these these datapoints are scattered across feature space and you want to fit a line or hyperplane to create a linear seperation boundary of the circles and squares. That way you can simply classify all the points that fall above the line as circles and all that fall below as squares. The issue here arises from the novel topic of infinite possiblities in space. There's no easy way to determine which is the optimal position for this line to be in, because you could slightly change its position infinitely and basically accomplish the same thing. There needs to be a more concrete set of rules for this algorithm to be commputationally efficient and just generally effective at its task.

The answer to this dilemma is the margin. The margin is essentially the maximum space between the hyperplane and its closest data point(s) in both directions. The datapoints which have the minimum distance from the hyperplane therefore define the margins border and are therefore termed `support vectors`. The reason we want to maximize this margin is because the larger it is, the more confident we can be in the accuracy of our classifications based on which side of the hyperplane data points are on.

The next limitation that arises from this point is what happens when the data is not linearly seperable, meaning there's no clearly defined hyperplane to seperate classes without there being missclassifications. One way of making the algorithm less strict when it comes to linear seperation boundaries is adding a slack variable to the objective function (described below) to correct the objective function so that it satisfies its constraints. This concept is called a hard/soft margin. A hard margin as its name implies is strict and allows for no slack variables, while a soft margin allows for the use of slack variables to make the decision boundary less strict but imposes bias by making the fit less accurate to the actual data. The use of slack variable is controlled

by a `C` variable which reperesents the budget for adding slack variables. If C is very low, larger slack variables will be used while when the opposite is true, slack variables will be more highly penalized and will then shrink. This penalty term concept is similar to the ideas behind `L2` regularization.

Another interesting way to solve more extreme situations, where data is totally not linearly seperable is `SVM kernels`. A kernel in this context is an interesting mathematical hack used to make two dimensional data, that's not linearly seperable in two dimensions linearly seperable in higher dimensional space by engineering a new variable to add that more dimentions to the data, then finding the linear seperation hyperplane in three dimensional space and finally projecting it back down to 2D space.

I won't delve too deep into the mathematics here, but I will enumerate and summarize the different variations of the kernels that can be applied to SVM.

Sources

https://www.geeksforgeeks.org/advantages-and-disadvantages-of-logistic-regression/

https://www.analyticsvidhya.com/blog/2021/04/forward-feature-selection-and-its-implementation/

https://towardsdatascience.com/understanding-random-forest-58381e0602d2