Logistic Regression

One of the most used classification techniques used in Machine Learning is Logistic Regression, as a lot of problems to be solved have a dichotomous outcome like the classification of a tumor being benign or malign and predicting if a student will pass an exam or not. This is called Binary Logistic Regression. On the other hand, Logistic Regression is also used in problems that have non-dichotomous outcomes, such as the prediction of weather, where outcomes include a day being sunny, cloudy, rainy, stormy etc… This multi-class type of Logistic Regression is often called Multinomial Logistic Regression. In this document, Binary Logistic Regression will be used as the tackled problem has a dichotomous outcome, where it is either 0 or 1.

In Logistic Regression instead of fitting a line of best fit as a hypothesis like in Linear Regression, the sigmoid function is used to form a hypothesis as seen in Figure ??, which gives the estimated probability of the outcome y being 1, given an input x with parameter θ.

$$h_\theta(x) = g(\theta^T x)$$

$$g(z) = \frac{1}{(1+e^{-z})}$$

$$h_\theta(x) = P(y=1|x;\theta)$$

Therefore the hypothesis will give a value from 0 to 1

$$0 \geq h_\theta(x) \geq 1$$

In order to classify a certain outcome as either 0 or 1, a decision boundary needs to be implemented, where if $h_\theta(x) \geq \frac{1}{n}$, then $y = 1$ and if $h_\theta(x) < 1 - \frac{1}{n}$, then $y = 0$, or vice-versa, where n can be any number.

For Logistic Regression to predict a certain outcome and having n features from x_0 to x_{n-1}, the parameters θ for each x are chosen by minimizing the function J(θ). T optimally minimize J(θ) a convex function for parameters θ needs to be used, so that J(θ) doesn't have any local minima. Having one or more local minima proves problematic when applying algorithms to minimize J(θ). The Cost Function used in J(θ) for Logistic Regression to calculate the loss from the predicted outcome $h_\theta(x)$ and actual outcome y, is called the Cross-Entropy or Log-Loss Cost Function. The Cross-Entropy cost function is technically two separate Cost Functions using the logarithm of the hypothesis $h_\theta(x)$, which results in a smooth convex function for both y = 0 and y = 1.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\begin{aligned} \mathrm{Cost}(h_\theta(x), y) &= -\log(h_\theta(x)) && \text{if } y = 1 \\ \mathrm{Cost}(h_\theta(x), y) &= -\log(1 - h_\theta(x)) && \text{if } y = 0 \end{aligned}$$

By combining both cost functions, J(θ) can be simplified as follows:

$$J(\theta) = -\tfrac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))]$$

In order to minimize J(θ), various techniques like Gradient Descent, Newton-Raphson, Conjugate Gradient, BFGS and L-BFGS. The simplest algorithm is Gradient Descent, whereas other algorithms are often faster than Gradient Descent they are far more complex to implement.

What Gradient Descent does is it iterates through equation ?? to update θ_j until it converges. Θ_j is subtracted with the scaled partial derivative of J(θ) with respect to θ_j. The partial derivative is scaled with learning rate parameter alpha, which can either be fixed or varied to control the gradient descent step rate for each iteration.

```
Repeat {
    $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$
}
```

Regularization

When fitting a model using a particular training dataset, the hypothesis can often have the undesirable effect of having either a high bias and low variance or low bias and high variance, meaning that the model can be under-fitting or over-fitting, respectively. If a model has high bias and low variance, therefore under-fitting, it means that the relationship between the features and the outcome isn't represented accurately. On the other hand if a model has low bias and high variance, therefore over-fitting, it means that the model is too complex and fits the training data-set very accurately.

One way to compensate for this, is to add a regularization/penalty term to J(θ) to decrease the complexity, where the most commonly used regularization methods being Lasso/L1, and Ridge/L2 regularization.

Lasso/L1 Regularization

In Lasso Regularization a penalty term which sums up the magnitude of all coefficients except the bias θ_0, and then scaled by parameter λ is added to J(θ) (reference to J(θ) equation above).

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Ridge/L2 Regularization

In Ridge Regression a penalty term which sums up the square of all coefficients except the bias θ_0, and then scaled by parameter λ is added to J(θ) (reference to J(θ) equation above).

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

For both regularization methods, if λ = 0, the penalty will be 0 as both regularization terms will equate to 0. The larger the λ parameter is the larger the penalty on those coefficients that add complexity to the model, therefore reducing the variance. The main difference between L1 and L2, is that L1 can make coefficients equal to 0, therefore eliminating features from the data-set, to reduce overfitting.

(For the regularization cost functions above use this format, so that it matches the cost function used in the Logistic Regression section, so that we can refer to it, when saying we added a term to it.)

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \boxed{\lambda \sum_{j=1}^{n} \theta_j^2}\right]$$

$$\min_\theta J(\theta)$$

Implementation of Logistic Regression

The Logistic Regression Implementation in Pure Python was done in the script logistic_regression.py. It is implemented as sci-kit learn compatible estimator class, which facilitates the integration of the implemented Logistic Regression, with scikit-learn functions that require certain estimator functions and attributes. This was done by inheriting the BaseEstimator and ClassifierMixin classes from scikit-learn, which provide the necessary functions like get_params, and set_params. In order to be compataible with scikit-learn's estimator standards, a fit() function which takes feature values X and label classes y, is used to fit the model based on X and y, while a predict() function which takes feature values X, is used to classify an outcome for the feature values X.

The fit() function uses Gradient Descent to minimize J(θ), where feature coefficients vector θ values are initialized to 0, and a bias vector initialized to 1 is added to each example's coefficients. It also has the option to either use L1 or L2 regularization.

The predict() function predicts the outcome of a particular company's set of data, to determine if it will go bankrupt or not, by passing the features through the already trained logistic model.

The estimator class' name is LogisticRegression and takes the following parameters;

- alpha, which specifies the learning rate for the fit() function's Gradient Descent algorithm.
- Threshold, ???? (delete this?)
- max_epoch, which specifies the maximum iterations for the fit() function's Gradient Descent algorithm.
- penalty, which specifies the fit() and __cost_function() functions' regularization method used, either L1 or L2 regularization, for both Gradient Descent and the Cost Function.
- lambda_t, which specifies the fit() and __cost_function() functions' regularization term lambda.
- Verbose, which is used in the fit() function to specify if the current loss and current epoch/iteration in Gradient Descent are to be displayed.

Apart from the fit() and predict() estimator class functions, it also includes the following functions:

- __sigmoid(z), which implements the sigmoid function, taking z as a paramater
- __criteria_met(), which checks if the maximum epoch/iterations specified has been reached.
- __cost_function(y, h, thetas), which implements the simplified cost function J(θ), taking the actual outcome *y*, estimated outcome *h* and the coefficients/weights *thetas*

The Logistic Regression estimator class was used in this project to rank features using Recursive Feature Elimination scikit-learn's built-in library. As per (https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html) and also other scikit-learn feature selection builtin functions, an estimator that has a *coef_* or *feature_importance_* in its fit() function is needed. This is because a scikit-learn feature selection function like RFE() has to use the *coef_* or *feature_importance_* attribute to access the object's features' coefficients to perform feature selection on.

Cross-Validation

Cross-Validation is a technique used to evaluate and validate a model’s performance. The main concept behind Cross-Validation is to partition the available data in to a training set, which is used to train the model, and a testing set, which is used to test the model. The advantage of doing so is to prevent bad practices like using the same dataset to both train and test the model, as this will not validate your model for use in other scenarios with a different dataset.

The most commonly used Cross-Validation technique is k-fold Cross-Validation, where the dataset available is split into k partitions, and k-1 partitions is used as a training set and the remaining partition is used as a testing set. Then this process is iterated for k number of times, so that every single partition is used as a testing set once. Then an accuracy metric is performed for each iteration/fold, and all the accuracy metrics are averaged out. This reduces both bias and variance as the original dataset is used for both training and testing sets, because the model is neither being overfitted due to having 1 large training set, nor is it being underfitted due to having a larger test set than the training set. The dataset prior to splitting can remain either contiguous or randomized.

K-fold Cross-Validation, therefore can be used for model selection, where multiple models are trained and tested using the same folds, and their accuracy metrics compared at the end. This enables the selection of the best performing model for that particular application, with the model having the least residuals/errors being the best.

A more optimized K-Fold Cross-Validation technique is, the Stratified K-Fold Cross Validation Technique, which is particularly useful when having class imbalance. This ensures that each partition contains the same number of different classes, so that the data in each partition is a complete set, that describes all possible outcomes. Another solution to tackle this data imbalance problems is to still use the contiguous or randomized k-fold technique, but prior to performing k-fold splits, the data is oversampled so that the class imbalance is eliminated. The latter technique which uses oversampled data is the one used in this project.

Feature Selection and Dimensionality Reduction

Feature Selection techniques are methods used to eliminate features which may be correlated to other important features, therefore eliminating redundancy, which in turn simplifies the hypothesis. By simplifying the hypothesis, both performance and memory allocation are improved while also reducing the chance of overfitting. Dimensionality Reduction Techniques achieves the same goal, but with a different approach. It differs from Feature Selection as most of the information in the dataset is kept and is instead compressed rather then selecting features to eliminate. This is done by reducing multiple correlated features/dimensions into fewer synthetic features/dimensions, like compressing 2 dimensions/features into 1 dimension/feature, or similarly, 3 dimensions/features into 2 dimensions/features.z

Recursive Feature Elimination

This Feature Selection technique, as the name suggests, recursively fits the model and eliminates the least important feature with every iteration. This is done by ranking the features according to their coefficient weight and eliminating the least weighted feature/s, depending if the algorithm is set to eliminate 1 or more feature with every iteration. After each iteration the model is fitted again, and again the lest weighted feature/s are eliminated, until the specified number of maximum iterations or features to be eliminated is reached. In this project, this will be used with the Logistic Regression classifier.

Chi^2 Feature Selection

The Chi^2 test in itself is used to determine if two events are dependent or not. It is often used in Machine Learning to rank features based on their Chi^2 score, related to the outcome. Features which are not found to be dependent on the outcome, are discarded. Usually a specified number of features to be selected is provided so that only the top ranked features are selected.

Principal Component Analysis

Principal Component Analysis, or PCA for short, is a Dimensionality Reduction Technique where its main objective, as the name suggests, is to identify principal components in a dataset. Principal Components are identified with the help of a covariance matrix Σ of the dataset. When two or more variables have a high positive covariance, the information in both can be represented by a single Principal Component.

$$\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T$$

After calculating the covariance matrix, the eigenvectors and their eigenvalues are calculated by decomposing the covariance matrix. This is done by several methods, but the most commonly used method is Singular-Value Decomposition, or SVD for short. What SVD does is decompose an $m \times n$ matrix into 3 matrices, an $m \times n$ unitary matrix U , an $m \times n$ diagonal matrix S and an $n \times n$ unitary matrix V . From this decomposition, the eigenvalues and eigenvectors are extracted.

$$\Sigma = U \cdot S \cdot V^T$$

The eigenvectors represent the direction of the new axis’ and the eigenvalues represent the variance of data on that particular eigenvector. The eigenvectors are then sorted by their eigenvalues, and the eigenvectors that have the least variance can be discarded. For a k-dimensional dataset, there can only be k-eigenvectors, as all the combination of eigenvectors has to span over the k-dimensional space. Therefore when eliminating the eigenvectors that represent the data with least variance, dimensions are being reduced, as their data can still be projected with minimal error on the eigenvectors that represent the data with most variance.

Rescaling and Normalization

Normalization

Normalization is a preprocessing step needed for the majority of algorithms in Machine Learning and Statistics to work or improve performance, such as Principal Component Analysis when using a correlation matrix and Chi^2 algorithms, and also for Logistic Regression when using Graident Descent, as it makes minimizing the cost function more efficient. Normalization rescales the numeric data within a specified range from x_{min} to x_{max}, more commonly 0.0 to 1.0, and provides a more clear boundary for specific algorithms to work with and also for data that doesn’t have a normal distribution.

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Another solution instead of the one presented in eq???, is to use linear interpolation, where the data x is scaled within x_{min} and x_{max}.

Standardization

Standardization works well when all of the data is normally distributed or in algorithms like Principal Component Analysis when using a covariance matrix or when the data needs to represent the spread/variance. The resulted data will have a mean of 0 and standard deviation of 1. Standardization differs mainly from normalization as the new data isn’t bound by a range.

$$x_{new} = \frac{x - \mu}{\sigma}$$

Imputation

Mean

Mean Imputation imputes the missing data with the mean of that particular feature, where the data can only be numeric. Due to the nature of all the missing data being imputed with the same value, it sometimes has the undesirable effect of adding bias and reducing variance, which in turn affects the correlation value between other features.

Mode

Mode Imputation imputes the missing data with the most frequent value for that feature, with the intuition that the modal value has a higher chance of being in the dataset. The advantage of this technique is that it can also be used on non-numeric data.

Assumptions of Logistic Regression

Binary Logistic Regression assumes that the outcome is dichotomous or in other words has two possible outcomes, whereas Multinomial Logistic Regression assumes that there are more than two possible outcomes. It also assumes that the dataset contributes to the model not being overfitting, nor underfitting. This means that a balance between bias and variance is assumed, with means of feature selection and dimensionality reduction in case of high variance, and gathering more meaningful data, in case of high bias. Logistic Regression also assumes that all training examples are independent of each other, therefore no colinearity is present between the training examples. As it’s also the case for any classifier, enough examples to represent the minority class’ distribution should be present so that the model can be trained to predict that particular outcome, as more patterns for that outcome have been present in the training data.

Assumptions of Decision Trees

Random Forest

Random forest is an ensemble of Decision Trees, which were described in subsection 2.1.1. This technique utilizes multiple Decision Trees (weak learners) to average out the predictions made by each tree, and this is done to reduce overfitting and to reduce the low bias, high variance trade-off found in a Decision Tree. There are many variations for this implementation such as Bagging and Boosting.

Random Forests by their default nature, use a technique similar to Bagging, which is essentially feature bagging for each Decision Tree, instead of the dataset. This means that only a particular number, or percentage of random features, are selected in each split of each Decision Tree, as stronger features can be left out in a number of decision trees. In the end the average of all the Decision Tree’s outcomes is taken. Random Forests, still use data bagging for each Decision Tree to increase the accuracy further, as the randomness decreases the correlation between the Decision Trees. Several methods to determine the selection of features for best split exist. The most simple ones are to specify an integer real number of a percentage of the total features. The most used methods are to take the square root and base 2 logarithm of the total number of features, to further decrease the correlation between Decision Trees.

Another technique called Boosting is used in Random Forests, where each Decision Trees are built iteratively. This is because in order for the next Decision Tree to be built, information about the previous Decision Tree should be present. So instead of averaging the classification results as in Bagging, Decision Trees are iterativley built based on also iterativley updated data and added together to create one model. The first iteration takes a randomly bagged data as in non-boosted Random Forests to build the first Decision Tree, but the same training data is used for testing on the newly built Decision Tree. When building the next Decision Tree, another random bag of data is taken, but taking into consideration the examples that had the most errors in their outcome from the test done on the previous Decision Tree. These examples are more likely to get selected for the current dataset to build the next Decision Tree. After the second Decision Tree is built, the same dataset used for training it used for testing on the current Decision Tree. The outputs of the first and second Decision Tree are then added together, where it might be found that previous error might have been improved, but new errors are now introduced. The same process is then repeated again to build the next Decision Trees, so that a stronger final model is built at the end, with every iteration correcting the newly introduced errors.