Model Selection

Introduction

The central issue in all of Machine Learning is "how do we extrapolate what has been learnt from a finite amount of data to all possible inputs 'of the same kind?". We build models from some training data. However, the training data is always finite. On the other hand, the model is expected to have learnt 'enough' about the entire domain from where the data points can possibly come. Clearly in almost all realistic scenarios the domain is infinitely large. How do we ensure our model is as good as we think it is based on its performance on the training data, even when we apply it on the infinitely many data points that the model has never 'seen' (been trained on)? This module will introduce you to some of the perspectives on this question and its implications.

Occam's Razor

A predictive model must be as simple as possible, but no simpler. Often referred to as the **Occam's Razor**, this is not just a convenience but a fundamental tenet of all of machine learning.

Before we explore this further, let's first get some intuitive understanding of what it means for a model to be 'simple'. To measure the simplicity, we often use its complementary notion — that of the complexity of a model. More complex the model, less simple it is. There is no universal definition for the complexity of a model used in machine learning. However here are a few typical ways of looking the complexity of a model.

1. Number of parameters required to specify the model completely. For example in a simple linear regression for the response attribute y on the explanatory attributes x_1, x_2, x_3 the model $y = ax_1 + bx_2$ is 'simpler' than the model $y = ax_1 + bx_2 + cx_3$ — the latter requires 3 parameters compared to the 2 required for the first model.

2. The *degree* of the function, if it is a *polynomial*. Considering regression again, the model $y = ax^2 + bx^3_1$ would$_2$ be a more complex model because it is a polynomial of degree 3.

3. Size of the best-possible representation of the model. For instance, the number of bits in a binary encoding of the model. For instance, more complex (messy, too many bits of precision, large numbers, etc.) the coefficients in the model, more complex it is. For example, the expression $(0.552984567 * x^2 + 932.4710001276)$ could be considered to be more 'complex' than say $(2x + 3x^2 + 1)$, though the latter has more terms in it.

4. The depth or size of a decision tree.

Intuitively more complex the model, more 'assumptions' it entails. Occam's Razor is there- fore a simple thumb rule — given two models that show similar 'performance' in the finite training or test data, we should pick the one that makes fewer assumptions about the data that is yet to be seen. That essentially means we need to pick the 'simpler' of the two models. In general, among the 'best performing' models on the available data, we pick the one that makes fewest assumptions, equivalently the simplest among them. There is a rather deep relationship be- tween the complexity of a model and its usefulness in a learning context. We elaborate on this relationship below.

1. **Simpler models are usually more 'generic'** and are more widely applicable (are generalizable). One who understands a few basic principles of a subject (simple model) well, is better equipped to solve any new unfamiliar problem than someone who has memorized an entire 'guidebook' with several solved examples (complex model). The latter student may be able to solve any problem extremely quickly as long as it looks similar to one of the solved problems in the guidebook. However given a new unfamiliar problem that doesn't fall neatly into any of the 'templates' in the guidebook, the second student would be hard pressed to solve it than the one who understands the basic concepts well and is able to work his/her way up from first principles.

2. **Simpler models require fewer training samples** for effective training than the more com- plex ones and are consequently easier to train. In machine learning jargon, the **sample complexity** is lower for simpler models.

3. **Simpler models are more robust** — they are not as sensitive to the specifics of the training data set as their more complex counterparts are. Clearly, we are learning a 'concept' using a model and not really the training data itself. So ideally the model must be immune to the specifics of the training data provided and rather somehow pick out the essential characteristics of the phenomenon that is invariant across any training data set for the problem. So, it is generally better for a model to be not too sensitive to the specifics of the data set on which it has been trained. Complex models tend to change wildly with changes in the training data set. Again, using the machine learning jargon **simple models have low variance, high bias and complex models have low bias, high variance**. Here 'variance' refers to the variance in the model and 'bias' is the deviation from the expected, ideal behavior. This phenomenon is often referred to as the **bias-variance tradeoff**.

4. **Simpler models make more errors in the training set** — that's the price one pays for greater predictability. **Complex models lead to overfitting** — they work very well for the training samples, fail miserably when applied to other test samples.

Overfitting

Overfitting is a phenomenon where a model becomes way too complex than what is warranted for the task at hand and as a result suffers from bad generalization properties. Let's consider a couple of examples to illustrate this.

1. A trivial 'model' that would have 'learnt' perfectly from any given dataset is one that just memorizes the entire dataset. Clearly the error committed by such a model on the dataset which it was 'trained' on would be zero. However, it is clear that such a model can do nothing other than answer questions (though perfectly) about the dataset it was trained on. The 'model' will effectively be incapable of doing anything better than a random guess on test data point that is outside the training dataset. This would be an extreme example of the overfitting phenomenon — perfect on the training data but unacceptably large error on test data.
2. Consider a set of points of the form $(x,\ 2x + 55 + s)$ for various values of x, where s is a Gaussian noise with mean zero and variance 1. We can carry out a linear regression on this dataset, but the regression line is unlikely to pass through any of the points exactly. We can also compute a polynomial fit on this dataset that will pass through almost every one of the given points perfectly. See Figure 1 where a straight line fit and a polynomial fit are both used to interpolate for a wide range of values of x. It is obvious from the figure that the polynomial fit is wildly off the mark for x values that are not part of the ones used for the 'training' (these points are marked with a small circle in the figure). The polynomial model badly *overfits* the given data whereas the straight-line fit is likely to behave predictably for a large of values of x and y that follow a similar pattern.

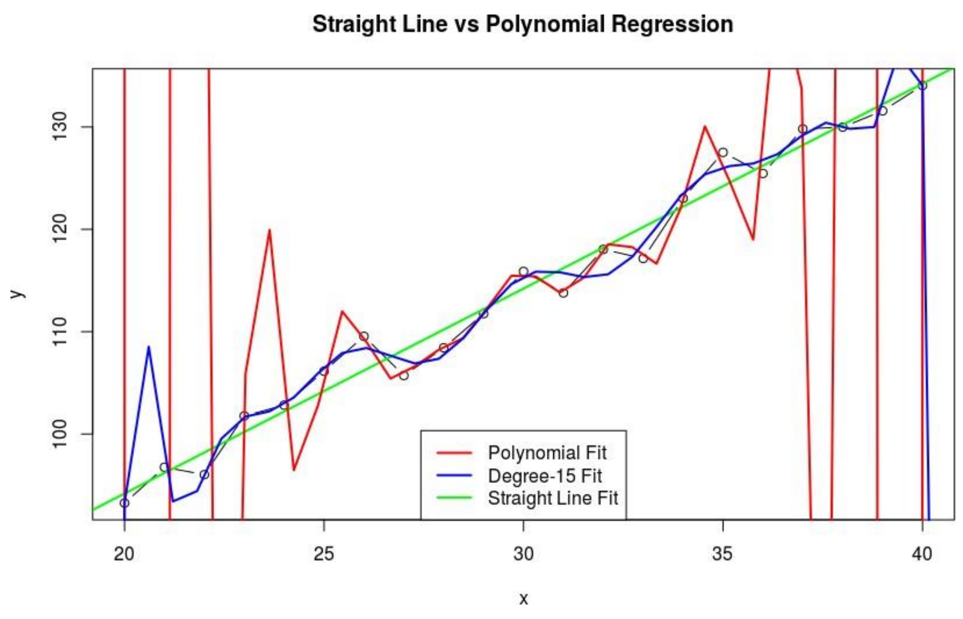


Figure 1: Polynomial vs Straight Line Fit

In short, overfitting makes a model become too specific to the data it is trained on and fails to generalise to other unseen data points in the larger domain. A model that has become too specific to a training dataset has actually 'learnt' not just the hidden patterns in the data but also the noise and the inconsistencies in the data. In a typical case of overfitting, the model performs very well on the training data but fails miserably on the test data.

Techniques to tackle overfitting:

1. Apply regularization techniques to reduce model complexity.
2. Increase the training data
3. Reduce the number of insignificant features by feature selection
4. Apply cross validation schemes to train multiple models on the same dataset
5. Early stopping to prevent the model from memorizing the dataset

Underfitting is a phenomenon where a model is too naive to capture the underlying patterns/trends of the data. Such a model does not learn enough from the training data and hence fails to generalize on new data which leads to unreliable results. Underfitting usually happens when you have less training data to build a sufficiently complex model. In a typical case of underfitting, a model fails to perform well on the training data as well as the test data.

Techniques to tackle underfitting:

1. Increase the training data
2. Increase the number of features by feature engineering
3. Increase the model complexity by adding more parameters
4. Increase the training time until the objective function is minimized

The image below clearly depicts the cases of underfitting and overfitting. The model in the first graph does not fit the data points at all and shows underfitting. The third graph overly captures the patterns in the data and seems like memorizing it. The second graph shows a right balance between the other two models and is neither underfitting nor overfitting. Thus, the goal lies in building a model which strikes the right balance and fits the data points sufficiently well for a good model.

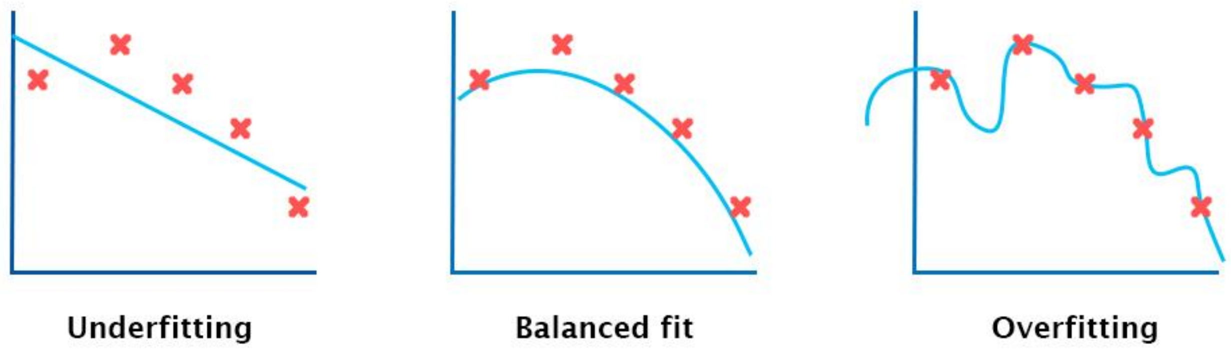
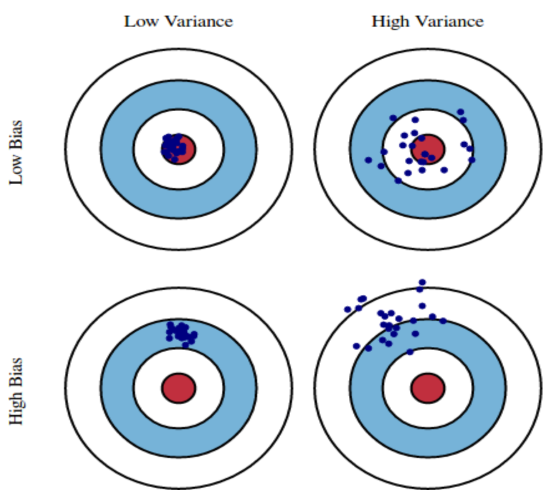


Figure 2: Model Fitting

Bias-Variance Trade-off

Consider the example of a model memorizing the entire training dataset, that we considered earlier. Clearly this 'model' will need to change for every little change in the dataset. The model is therefore very unstable and extremely sensitive to any changes in the training data. However, 'simpler' model that abstracts out some pattern followed by the data points given is unlikely to change wildly even if more points are added, points are removed or if some of the given points are perturbed a little. The 'variance' of a model is the variance in its output on some test data with respect to changes in the training dataset. In other words, *variance* here refers to the degree of changes in the model itself with respect to changes in the training data.

Bias quantifies how accurate is the model likely to be on future (test) data. Complex models, assuming you have enough training data available, can do a very accurate job of prediction. Models that are too naive, are very likely to do badly. Again, a trivial example of an utterly naïve model is one that gives the same answer to all test inputs — makes no discrimination whatso- ever, and just repeats the same answer no matter what the question is!! This model will indeed have a very large bias — its expected error across test inputs is very high.



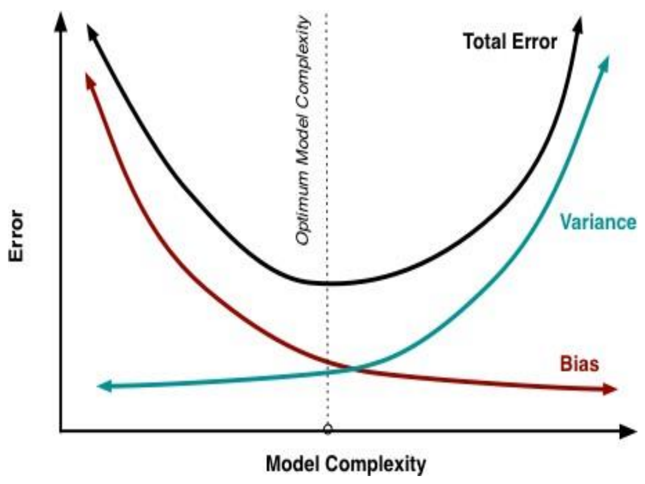


Figure 3: (Left) Illustration of Bias and Variance; (Right) Bias-Variance Tradeoff

The left-hand side of Figure 2 illustrates Bias and Variance using a target shooting analogy. A 'model' whose shots are clustered together is one that has a small variance and one whose shots are close to the "bull's eye" has a small bias. A 'consistent' shooter will have a small variance and a 'good' shooter will have a small bias. So, in other words variance is about consistency and bias is about accuracy.

The right-hand side of Figure 2 illustrates the typical tradeoff between Bias and Variance — low complexity models have high bias and low variance and high complexity models have low bias but high variance. Notice that our naive 'constant' model will never change and hence its variance is zero, though the bias is high. The 'best' model for a task is one that balances both — achieves reasonable degree of predictability (low variance) without compromising too much on the accuracy (bias).

Regularization

Regularization is the process used in machine learning to deliberately simplify models. Through regularization the algorithm designer tries to strike the delicate balance between keeping the model simple yet not making it too naive to be of any use. For the data analyst, this translates into the question "How much error am I willing to tolerate during training, to gain generalizability?"

The simplification can happen at several levels — choice of simpler functions, keeping the number of model parameters small, if the family is that of polynomials then keeping the degree of the polynomial low, etc. These are done by the designer and is typically not referred to as regularization. Regularization is the simplification done by the training algorithm to control the model complexity. Some typical regularization steps for various classes of ML algorithms include:

1. For Regression this involves adding a regularization term to the cost that adds up the absolute values or the squares of the parameters of the model.
2. For decision trees this could mean 'pruning' the tree to control its depth and/or size.

We will explore the specifics of these regularization strategies when we cover these algorithms in detail later in the course.

Evaluation Metrics for Classification and Regression

How do you choose between different models for a given business problem? This is done by comparing the performance of these models using different evaluation metrics. Let's now understand the basics of different evaluation metrics available for both classification and regression problems.

Classification Metrics

Some of the classification metrics include **accuracy, precision, recall, confusion matrix, F1 score, and the AUC-ROC score**.

Accuracy: Consider a dataset which is highly imbalanced, with 99.83% of the observations being labelled as non-fraudulent transactions, and only 0.17% of observations being labelled as fraudulent. So, without handling the imbalances present, the model overfits on the training data and is, therefore, classifying every transaction as non-fraudulent and hence, achieving the aforementioned accuracy. Therefore, accuracy is not always the correct metric for solving classification problems.

A high value of accuracy in the case of a class imbalance problem makes it difficult to understand whether or not it is the true representation of the predictive power of a model. This means that you need to look out for other metrics that can handle the minority class predictions in a better manner. Accuracy is given by:

$$Accuracy = \frac{True\ Positives + True\ Negatives}{Total\ Number\ of\ Predictions}$$

It gives equal importance to both false positives and false negatives. However, this may not be the case in most business problems. Accuracy is preferred in cases where the cost of predicting false positives and false negatives is roughly the same. For example, suppose you want to predict an image whether it is a cat or not. In this case, both the cost of false positives and the cost of false negatives do not exceed one another, and you will not be charged any penalty or additional costs if any kind of misclassification occurs in this case.

Precision is given by:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

Precision is preferred in cases where the cost of false positives is much higher than that of false negatives. For example, suppose you are trying to predict whether an email is spam or ham. If an email is classified as spam, then it will be moved into the trash folder; this is the case of a false positive. On the other hand, in case of false negatives, the spam email stays in the inbox. So, the potential problem in this case is that an email that was incorrectly classified as spam would have contained some important information, and the cost of missing that information is much higher than having a spam email in your inbox.

Recall is given by:

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

Recall is preferred in cases where the cost of false negatives is higher than that of false positives. For example, suppose you are trying to predict whether a person has Covid-19 or not. If an actual Covid-19 positive patient is predicted negative, and this is the case of false negative for which the cost is quite high as they are a potential carrier of the virus, then whoever comes in contact with the patient would get infected.

AUC-ROC: It is used to understand the strength of the model by evaluating the performance of the model at all the classification thresholds. ROC curve is a plot between the True Positive Rates and the False Positive Rates. A default threshold of 0.5 as in confusion matrix, precision, recall and F1 score is not always the ideal threshold to find the best classification label of the test point. Because the ROC curve is measured at all thresholds, the best threshold would be one at which the TPR is high and FPR is low, i.e., misclassifications are low. Higher the area under the ROC curve, better is the model. This is known as the AUC-ROC score.

F1 score: After determining the optimal threshold, you can calculate the F1 score of the classifier to measure the precision and recall at the selected threshold. When you increase the recall, you will also decrease the precision. In cases where both the precision and the recall need to be optimised, the F1 score would be the best option. It is calculated as the harmonic mean of both the precision and the recall.
The F1 score is given by:

$$F - measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

In some cases, there will be a trade-off between the precision and the recall. In such cases, the F-measure will drop. It will be high when both the precision and the recall are high. Hence, depending on the business case at hand and the goal of data analytics, you need to select an appropriate metric.

Regression Metrics
Some of the evaluation metrics for a regression task include:

Root Mean Square Error(RMSE):

RMSE is the most commonly used regression metrics. It represents the sample standard deviation between the actual and the predicted values. It is calculated using the formula: Some of the evaluation metrics for a regression task include:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y})^2}$$

Where yi and y^ are the actual and predicted values respectively and n is the number of observations.

Mean Absolute Error(MAE):

MAE is the average of the absolute difference between the actual and the predicted values.It is calculated using the formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} | \, y_i - \hat{y}|$$

Where yi and y^ are the actual and predicted values respectively and n is the number of observations.

Mean Absolute Percentage Error(MAPE):

MAPE is the percentage equivalent of Mean Absolute Error. It can be defined as the percentage ratio of the residue over the actual value. It is in the form of percentage which makes it easier for people to comprehend, hence it is also a commonly used metric. It is calculated using the formula:

$$MAPE = \frac{100}{n}\sum_{i=1}^{n} \frac{|y_i - \hat{y}|}{y_i}$$

Where yi and y^ are the actual and predicted values respectively and n is the number of observations.

R-Squared (R^2)

R^2 also known as the coefficient of determination measures the proportion of variation in the dependent variable that is explained by all the independent variables in the model. Higher the value of R^2 better is the model. It is calculated using the formula:

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

Where $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y})^2$ is the Mean Square Error(MSE) and $\frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2$ is the variance in the y values.

Adjusted R-Squared

Adjusted R^2 is the modified version of R^2 and it adjusts for the number of independent variables in a model. It is a proportion of variation that is explained by the independent variables actually affecting the dependent variable. It is calculated using the formula:

$$R^2_{adj} = 1 - \frac{(1-R^2)(n-k)}{n-k-1}$$

Where n is the total number of observations and k is the number of independent variables in the model

The value of adjusted R^2 is always less than or equal to the R^2 value. R^2 keeps increasing on the addition of independent variables into the model irrespective of how well they are correlated with the dependent variable whereas adjusted R^2 penalizes the model for the inclusion of insignificant variables which do not help in improving the existing model. Hence, it is recommended to assess the performance of a multiple linear regression model by using only adjusted R^2 instead of a R^2 value because it is quite misleading.

Model Evaluation

The discussion has so far been on the choice of a class of models (linear, quadratic, etc.); the learning algorithm (like say linear regression) will then go ahead and find the best among this class of models to fit the given data. In this section we will discuss how we evaluate a specific model. Remember that evaluating how a model performs on the data it has been trained on is relatively straightforward. What we need to estimate however, to assess the usefulness of the model, is to evaluate its performance on data that it hasn't seen yet. Often, we will need to compare two models and finally choose one, rather than evaluating how good a model is in absolute terms.

We will first look at a couple of generic strategies for making best use of available data in or- der to arrive at a model that will generalize reasonably well. Of course, in a situation where the available data is in abundance, this is hardly a concern. However when data limited, which is often the case, notice that we are dealing with a fundamental dilemma — we need the training set to be as large as possible (small training set leaves us with the danger of missing out on crucial pieces of information required for the learning) yet we need a way to estimate the reliability of the model in test scenarios not covered by the training data. In both the methodologies — *Hold-Out Strategy* and *Cross Validation* — the basic idea is the same: to keep aside some data that will not in any way influence the model building. The part of the data that is kept aside is then used as a 'proxy' for the unknown (as far as the model we have built is concerned) test data on which we want to estimate the performance of the model. In the cross-validation technique, you split the data into train and test sets and train multiple models by sampling the train set. Finally, you can use the test set to test the final model once.

Specifically, you can apply the k-fold cross-validation technique, where you divide the training data into k-folds/groups of samples. If k = 5, you use k-1 folds to build the model and test it on the kth fold.

It is important to remember that k-fold cross-validation is only applied on the train data. The test data is used for the final evaluation. One extra step that we perform in order to execute cross-validation is that we divide the train data itself into train and test (or validation) data and keep changing it across "k" no. of folds so that the model is more generalized.

There are other cross validations schemes apart from k-fold cross validation which help in effective model evaluation. These are stratified k-fold and leave one out validation schemes. **k-fold cross-validation** is one of the most commonly used cross-validation schemes for both regression and classification problems, but for classification problems with class imbalance, you use another cross-validation scheme called stratified k-Fold, which ensures that the relative class proportion is approximately preserved in each train and validation fold. It is important in cases when there is a huge class imbalance (e.g., 98% good customers, 2% bad customers).

Leave One Out (LOO) is useful when you have a limited data set. It takes each data point as the 'test sample' once and trains the model on the remaining n - 1 data points. Thus, it trains a total of n models. One of the advantages of this scheme is that it utilizes the data well, since each model is trained on n - 1 samples. However, this scheme also has certain disadvantages. It is computationally expensive and is not efficient in tackling the problem of overfitting.

We also look at *hyperparameters* in this context. Refer Figure 3. A typical Machine Learning scenario is where we have a data source (possibly implicit) and an underlying system that we are trying to mimic.

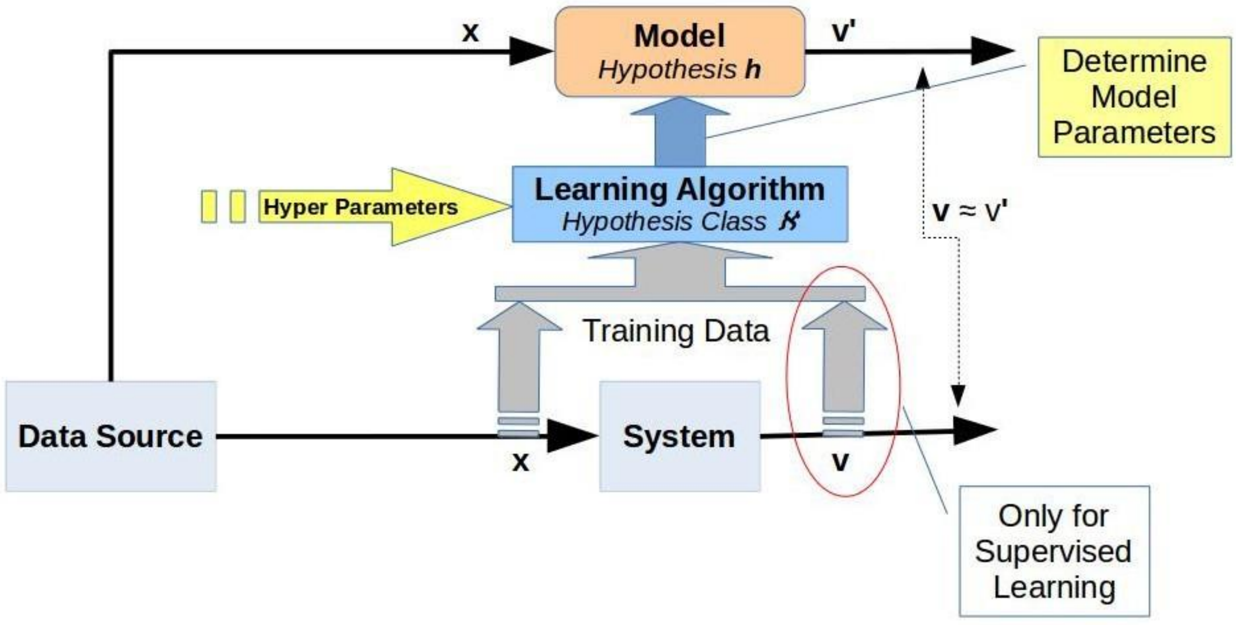


Figure 4: Machine Learning — A Broad Framework

For example, the data source could be an email repository or an email server. The 'system' is a human gatekeeper who is trying to segregate spam emails from the rest. The machine learning model we are trying to build is essentially trying to mimic a human 'system' of discriminating between spam emails and the rest. There is a *Learning Algorithm* that takes samples from the data source (this will constitute the training data) and the expected responses from the 'system' and comes up with a *model* that behaves a lot like the system we are trying to mimic. The model will often be used to predict what the system would have presumably done on any new test input that is given. In the figure, the inputs are denoted by the vector x, the expected 'system' output as v and model output as v^f. The learning algorithm works with a *hypothesis class* — each learning algorithm searches for the best possible model from a fixed class of models specific to the algorithm. For example, standard linear regression finds the best 'straight line' to fit the data, a decision tree algorithm finds the best decision tree to explain the dataset, a logistic regression model with a single explanatory variable looks for all models of the form $1/\ 1 + e^{-(b+a_1x)}$.

The output of the model is a specific hypothesis or model. For the above three examples, for a straight-line regression fit, the model is one specific straight line (with values for the coefficients) that fits the model best, for the logistic regression model we find specific values of a, b for the best fit, etc. *Hyperparameters* govern the way the learning algorithm produces the model.

Now what is the difference between a model parameter and a hyperparameter?

Model parameters are the parameters that are determined using the training set on which the model is built. These parameter values are not set manually but are decided by the machine while learning from the training data. These are then saved internally for the model to make predictions. Model parameters define the skill of your model for a given business problem.

Hyperparameters are simply the parameters that we pass on to the learning algorithm to control the training of the model and estimate the model parameters. Hyperparameters are choices that the algorithm designer makes to 'tune' the behaviour of the learning algorithm. The choice of hyperparameters, therefore, has a lot of bearing on the final model produced by the learning algorithm.

So anything that is passed on to the algorithm before it begins its training or learning process is a hyperparameter, i.e., these are the parameters that the user provides and not something that the algorithm learns on its own during the training process.

Hyperparameters are parameters that are passed on to the learning algorithm to control the complexity and performance of the final model. These are choices that the algorithm designer makes to 'tune' the behavior of the learning algorithm. The choice of hyperparameters, therefore, has a lot of bearing on the final model produced by the learning algorithm. They are a part of most of the learning algorithms which are used for training and regularization.

How do you know the right set of hyperparameter combinations for the model you are building? When you define a set of hyperparameter values to tune your model and check if it is optimum or not by evaluating its performance on the test set, the model is actually taking a sneak preview of the unseen data which is kept aside for the final model evaluation. The model is repeatedly exposed to the test set for examining each set of hyperparameter values before it is finalized for the actual test. This way of choosing the best set of hyperparameters for a model is not acceptable and this is where the notion of cross validation comes.

The original data is divided into three parts: train, validation and test sets. Each time a model is trained on the train set with a specific set of hyperparameters, this model is then validated and tuned on the validation set to figure out the optimum set of hyperparameter values and the model can revisit the validation set any number of times. After finalizing the model hyperparameters, a final evaluation of the model performance is done using the test set. The key thing to remember is that a model should never be evaluated on data it has already seen before.

Hyperparameter tuning is one of the essential tasks in the model building process. For a model to show the best performance on a given data, you need to tune its hyperparameter(s).

GridSearchCV() can be used to perform hyperparameter tuning using the cross-validation approach to find the optimum set of hyperparameters. The grid is a matrix that is populated on each iteration. Please refer to the image given below.

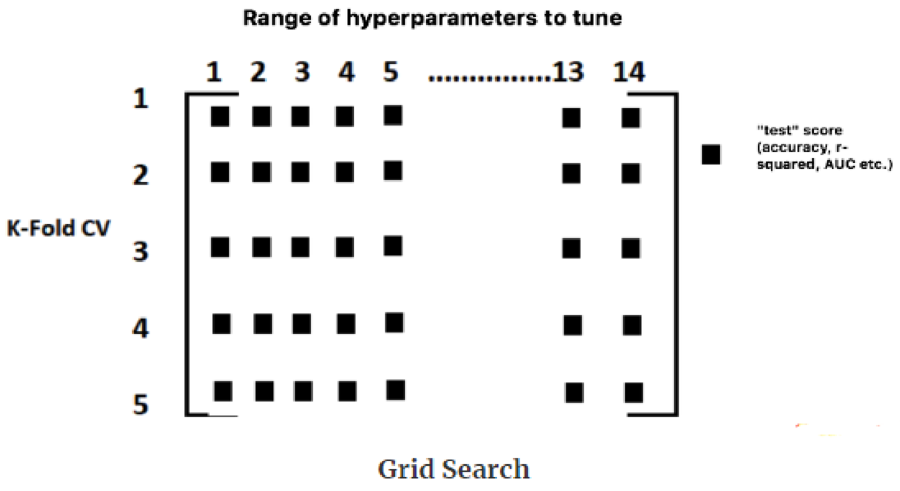

Range of hyperparameters to tune
1 2 3 4 5 ....13 14
1
2
3
4
5
K-Fold CV
"test" score (accuracy, r-squared, AUC etc.)
Grid Search

Figure 5: Grid Search Hyperparameter Tuning

In the figure, the value corresponding to each row in the grid represents the number of folds created for performing k-fold cross validation and the value corresponding to each column represents the different hyperparameter values that are used for building model to get the best results out of all possible combination of hyperparameter and cross validation fold values. You train the model using different hyperparameters each time and the estimated score is then populated in the grid.

There are broadly three steps that you perform while tuning the hyperparameter(s) in Python. These are as follows:

1. Create a cross-validation scheme: for example, how many splits do you want to set?
2. Specify the range of hyperparameters to tune
3. Perform grid search on the set range

But GridSearchCV is not helpful in case of large datasets. You cannot afford to keep running these codes for days in order to get the optimum results in case of larger data sets. Other efficient techniques of hyperparameter tuning are implemented in such cases and that is RandomizedSearchCV. It is a highly efficient technique that is used to identify the optimum set of hyperparameter values in fewer number of iterations. This technique performs quite well at a reduced cost and a shorter time for huge data sets and models with large numbers of hyperparameters. It does not perform an exhaustive search of hyperparameters over ranges where it does not find any merit; instead, it hops over a wide space in far less time as compared with GridSearchCV. It moves within the grid in a random fashion to find the optimum set of hyperparameters. RandomizedSearchCV produces almost the same result as GridSearchCV in just a matter of seconds.

End to End Modelling

In practice, you have many models to work with and you may get overwhelmed by the choice of algorithms available for handling classification and regression problems. But it is always advisable to **start with a simple model** depending on the type of problem that you have. Begin with a linear regression model if you have a regression task or a logistic regression model for a classification task. Using a simple model serves two purposes:

- It acts as a **baseline** (benchmark) model.
- It gives you an idea about the model performance.

Then, you may want to go for **decision trees** if interpretability is something that you are looking for and compare its performance with the linear/logistic regression model.

Finally, if you still do not meet the requirements, use **random forests**. But, keep in mind the **time and resource constraints you have,** as random forests are computationally expensive. Go ahead and build more complex models like random forests only if you are not satisfied with your current model and you have sufficient time and resources in hand.

But remember these are not the only models that you have. As you go ahead and explore new models, feel free to apply them in your business problems in order to get insightful and good performance results.

Feature Engineering

Feature engineering is an important step in any model building exercise. It is the process of creating new features from a given data set using the domain knowledge to leverage the predictive power of a machine learning model.

It plays a crucial role in extracting hidden insights and the underlying structure of data by modifying, combining, extracting or selecting features that are less prone to overfitting. Feature engineering is part of the data preparation stage and makes data more compatible with the algorithm. This, in turn, makes the subsequent stages of model building more effective and efficient.

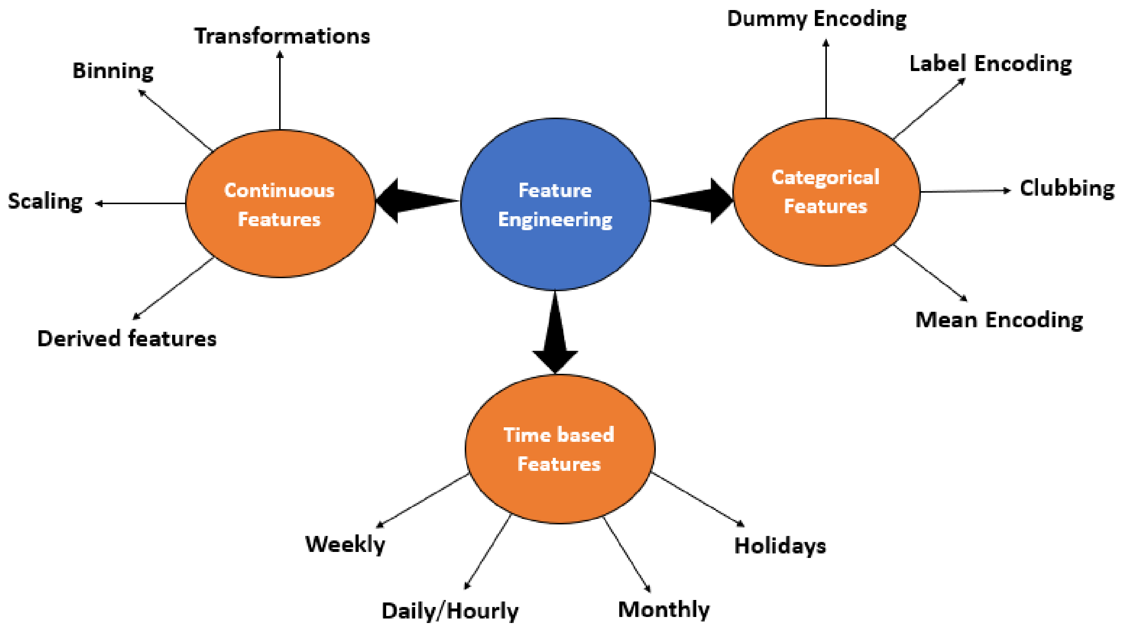


Figure 6: Feature Engineering

Some of the most commonly used techniques for handling **continuous numeric features** are as follows:

- Scaling (Standardization and Normalization)
- Binning
- Transformations (Log, Power transformations such as the Yeo Johnson transformation and the Box-Cox transformation)
- Derived features

Scaling is the technique of transforming features by standardizing or normalizing them to a given range/scale. It is performed to deal with highly varying values or units of different features in a data set at the same level. Scaling is most commonly performed in the following two ways:

Standardization is the process of rescaling feature values such that the resulting distribution has a mean value of 0 and a variance of 1. The formula for standardization is given below:

$$X_{new} = \frac{X_i - X_{mean}}{\sigma}$$

where X_{new} is the resultant new value, X_i is an observation, X_{mean} is the mean value of all the observations (denoted by X_i) and σ is the standard deviation.

Normalization is a technique of rescaling features within a specific range of [0, 1]. It can be done using the following formula:

$$X_{new} = \frac{X_i - Min(X)}{Max(X) - Min(X)}$$

where X_{new} is the resultant new value, X_i is an observation, min(X) and max(X) are the minimum and maximum values among all the observations (denoted by X_i), respectively.

There is no hard and fast rule regarding when to use standardization or normalization. The decision always depends on your business goals and the machine learning algorithm that you are trying to build.

An important point to remember is that you should always fit the scalar on the training data in order to avoid data leakage and the model's performance being compromised.

Binning is another technique of feature engineering in which continuous values are split into smaller groups/bins. Suppose you have a data set containing the age of different individuals along with other features. Now, transforming this age column into smaller bins of age groups, namely <25, 25–50 and >50, makes the data set more manageable and easier to understand, thereby allowing you to derive meaningful insights.

Log transformation is a technique used to transform highly skewed data to less skewed. It compresses the spread of the data on a higher scale, and hence, the shape of the data changes.

Power transformation is a family of transformations, among which the Yeo Johnson and Box-Cox transformations are the most popular.

In the **Box-Cox transformatio**n, the non-normal features are transformed into a normal distribution, which is an important assumption in many of the statistical techniques.

The **Yeo Johnson transformation** is somewhat similar to the Box-Cox transformation but does not require the input variables to be strictly positive.

Finally, the derived features are extracted from the already existing features by taking the ratios, products, sums, and so on.

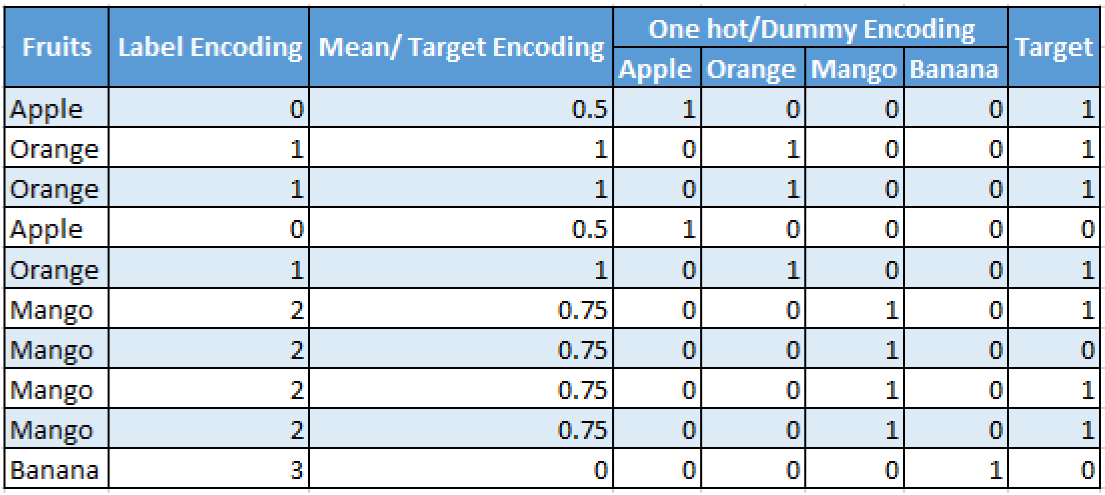
Some of the most commonly used techniques for handling **categorical features** are as follows:

- Dummy/One-hot encoding
- Label encoding
- Mean/target encoding
- Clubbing features (based on frequency and the target variable)

Dummy/one-hot encoding is the process of converting different levels of a categorical feature into vectors containing 1s and 0s, where 1 indicates the presence and 0 indicates the absence of a level or a category.

Label encoding is the process of converting each category in a column into numbers. This is done because many of the machine learning algorithms cannot process the features in categories. Therefore, before the model building part of the process, it becomes necessary to transform each level of category into numbers.

Mean/target encoding is the process in which the target variable is used to generate a new encoded feature. This is done by replacing a category with the mean value of the target variable corresponding to each occurrence of the same category. The following example will help you understand this better.



Fruits	Label Encoding	Mean/ Target Encoding	One hot/Dummy Encoding				Target
			Apple	Orange	Mango	Banana	
Apple	0	0.5	1	0	0	0	1
Orange	1	1	0	1	0	0	1
Orange	1	1	0	1	0	0	1
Apple	0	0.5	1	0	0	0	0
Orange	1	1	0	1	0	0	1
Mango	2	0.75	0	0	1	0	1
Mango	2	0.75	0	0	1	0	0
Mango	2	0.75	0	0	1	0	1
Mango	2	0.75	0	0	1	0	1
Banana	3	0	0	0	0	1	0

Figure 7: Categorical Feature Encoding

Here, different fruits are label-encoded by assigning numbers. But in the case of mean coding, as you can see in the case of mangoes, it occurs four times in this data, and the mean of the target variable corresponding to each occurrence of mango gives ¾ = 0.75. Hence, the new mean-encoded feature has a value of 0.75 wherever 'mango' occurs in the data. Also, one hot encoding creates four new feature columns containing 1s and 0s indicating the presence and absence of each category respectively.

Frequency encoding is a method of representing the different levels in a categorical feature using the frequency of their appearance in the data set. The one that appears most frequently in a data set is, in a way, given more weightage.

Apart from these, there are other techniques using which the different levels in a categorical column are clubbed based on their frequency of occurrence and on the target feature.

Some **time-based features** can be exploited as well in order to extract meaningful information such as weekly and hourly patterns, monthly cyclicity, holidays.

Class Imbalance

A machine learning algorithm works efficiently when there is an equal representation of each of the classes. However, if the data set contains a disproportionate number of records belonging to a particular class, then the prediction will be heavily biased. This is known as 'class imbalance' or a minority class problem.

An imbalanced class leads to **biased predictions** and **misleading accuracies**. When the majority class dominates over the minority class, the machine learning algorithm tries to learn the majority class better and fails to predict the minority class. The algorithm becomes incapable of identifying the minority class while making predictions. This results in misleading accuracies. As the model is predicting the majority class well, naturally, the accuracy score will be higher, and it will not capture the inefficiency of the model in predicting the minority class.

So, there are different techniques of handling class imbalance, which are as follows:

1. Random undersampling
2. Random oversampling
3. Synthetic minority oversampling technique (SMOTE)
4. Adaptive synthetic sampling method (ADASYN)
5. SMOTETomek (Oversampling followed by undersampling)

One key point to remember here is that always apply these resampling techniques on the train set, not on the test set. Applying them on the test set will lead to data leakage, and the model will remember the instances of the test set. Note that the true test is always on the unseen data. Hence, there will not be any unseen data left to test the model performance on. So, always remember to do the train-test split before applying class imbalance techniques.

There are various methods of mitigating the problem of class imbalance. They are as follows:

Random undersampling: In this method, you have the option of selecting fewer data points from the majority class for your model-building process. In case you have only 500 data points in the minority class, you will also have to take 500 data points from the majority class; this will make the classes somewhat balanced. However, in practice, this method is not effective because you will lose over 99% of the original data containing important information, which may lead to bias.

Tomek links is one of the undersampling techniques that is based on a distance measure. It removes unwanted overlaps between classes and majority class links are removed until all the minimally distanced nearest neighbour pairs are of the same class. It tries to find the nearest neighbours of the majority class and resample/remove most of the close nearest neighbours that overlap each other.

Random oversampling: Using this method, you can add more observations from the minority class by replication. Although this method does not add any new information, there is no information loss. It may also exaggerate the existing information to a certain extent, leading to the problem of overfitting.

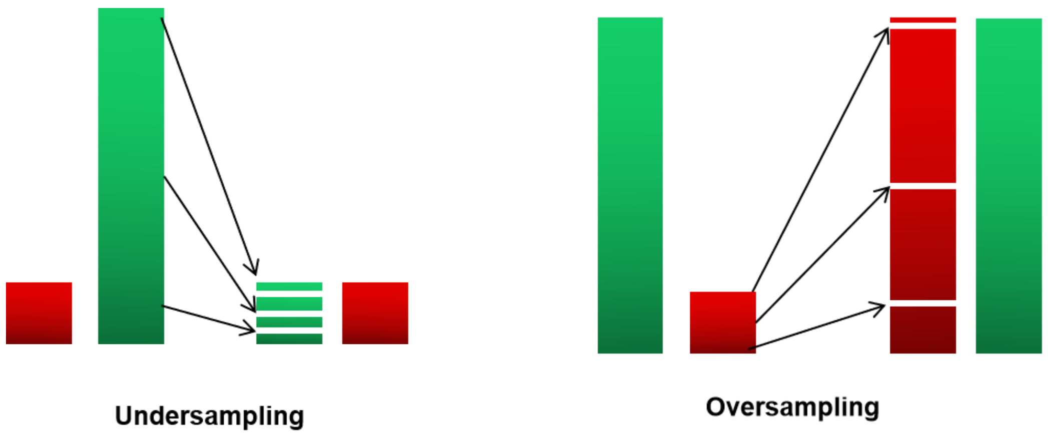


Figure 8: Undersampling and Oversampling

Synthetic minority oversampling technique (SMOTE): Using this technique, you can generate new data points that lie vectorially between two data points that belong to the minority class. These data points are randomly selected and then assigned to the minority class. This method uses the K-nearest neighbours to create random synthetic samples. The steps involved in this method are as follows:

1. Identifying the feature vector and its nearest neighbour
2. Taking the difference between the two
3. Multiplying the difference with a random number between 0 and 1
4. Identifying a new point on the line segment by adding the random number to the feature vector
5. Repeating the process for identified feature vectors

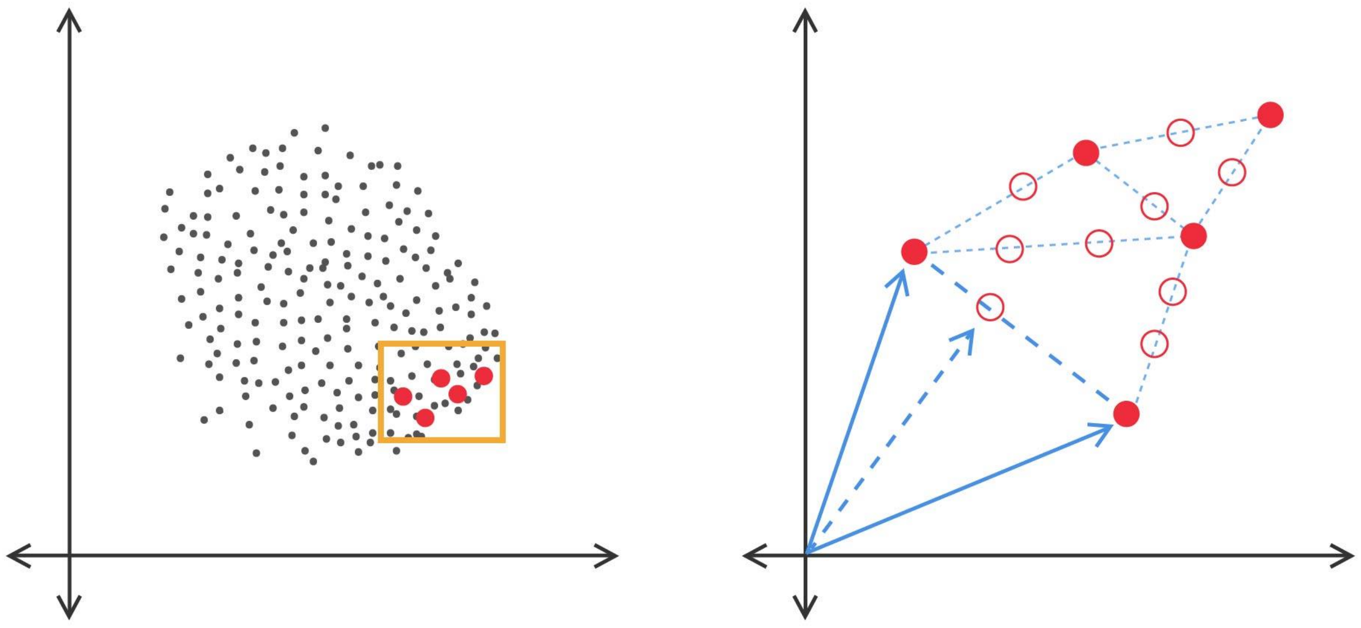


Figure 9: SMOTE(New synthetic sample points are added between the two homogenous class points.)

ADAptive SYNthetic (ADASYN): This method is similar to SMOTE, with a minor change in the generation of synthetic sample points for minority data points. For a particular data point, it will add a number of synthetic samples that will have a density distribution, whereas for SMOTE, the distribution will be uniform. Here, the aim is to create synthetic data for minority samples that are harder to learn, rather than the easier ones.

To sum it up, the ADASYN method offers the following advantages:

- It lowers the bias introduced by the class imbalance.
- It adaptively shifts the classification decision boundary towards difficult samples.

There is another approach that combines both the undersampling and the oversampling techniques. Tomek links and SMOTE use the undersampling and the oversampling techniques, respectively. Tomek links can be used as an undersampling method or a data cleaning method. So, Tomek links to the over-sampled training set as a data cleaning method. Thus, instead of just removing the majority class examples using Tomek links, SMOTE tries to increase the minority class to balance the data set.

It is important to understand that there is no single methodology that suits all problems. Let’s summarize all learnings related to imbalanced class below:

1. There are some cases where undersampling methods are performing better than oversampling methods and vice versa. However, it is important to use multiple methods for feature engineering of the data first and then compare the results to select the best possible method.
2. As undersampling methods remove the majority class from the data set, it can be an issue in many cases because you might lose significant information.
3. Oversampling methods generate samples from the data set, which can create irrelevant observations and also result in overfitting.

4. Interpreting and understanding the evaluation metrics and how they can be used to help solve a real-world business problem is extremely important.
5. Do not use accuracy score as a metric for model evaluation. In a data set with 96% non-claim observations, you will likely make correct predictions 96% of the times. A confusion matrix and precision/recall score are better metrics for evaluating the performance of a model.