Model Performance

Albert Mata

12/11/2018

Contents

Evaluating Model Performance

Machine learning algorithms have varying strengths and weaknesses and it's important to forecast how they will perform on future data. Predictive accuracy is not sufficient to measure performance, we need to use some other performance measures that reflect a model's ability to predict or forecast unseen cases.

Measuring performance for classification

Class imbalance problem: Trouble associated with data having a large majority of records belonging to a single class. A classifier predicting always *no defect* would be correct 99.99% of cases when a genetic defect is found in only 10 out of every 100,000 people.

The best measure for a classifier's performance is always the one that captures whether the classifier is successful at its intended purpose. We measure utility rather than raw accuracy. The goal of evaluating a classification model is to have a better understanding of how its performance will extrapolate to future cases. To achieve this goal, we have different types of data at our disposal:

- Actual class values (correct answers, solutions).
- Predicted class values (`predicted_outcome <- predict(model, test_data)`).
- Estimated probability of the predicition (if two models make the same number of mistakes but one is more capable of accurately assessing its uncertainty, then it is a smarter model).

We get these estimated probabilities of the predicition in R using `type = "prob"` or `type = "raw"` or similar as a parameter in `predict()`. This usually returns a probability (between 0 and 1) for each category of the outcome.

For convenience during the evaluation process, it's useful to construct a data frame containing the **a)** predicted class values, **b)** actual class values and **c)** estimated probabilities of interest. And we'll see that a model can be extremely confident and yet it can be extremely wrong. To answer if a model can still be useful in spite of its mistakes, we apply various error metrics to the evaluation data.

A closer look at confusion matrices

A **confusion matrix** is a table that categorizes predictions according to whether they match the actual value. Correct predictions fall on the main ↘ diagonal. Incorrect predictions are all the rest.

The most common performance measures consider the model's ability to discern one class versus all others. The class of interest is known as the **positive** class, whie all others are known as **negative**. This leads to predictions falling into one of these four categories:

- **True Positive (TP)**: Correctly classified as the class of interest.
- **True Negative (TN)**: Correctly classified as not the class of interest.

- **False Positive (FP)**: Incorrectly classified as the class of interest.
- **False Negative (FN)**: Incorrectly classified as not the class of interest.

And based on this we can create a 2×2 confusion matrix based on positive/negative actual/predicted values.

Using confusion matrices to measure performance

Prediction accuracy (success rate) vs **error rate**:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$error\ rate = \frac{FP + FN}{TP + TN + FP + FN} = 1 - accuracy$$

In R, we can easily tabulate a classifier's predictions into a confusion matrix using `table(df$actual_type, df$predict_type)` or similar `CrossTable` function from `gmodels` package to get some extra additional detail.

Beyond accuracy - other measures of performance

The **C**lassification **a**nd **Re**gression **T**raining package (`caret`) provides a large number of tools to prepare, train, evaluate and visualize machine learning models and data and includes functions to compute many performance measures.

We can create confusion matrices indicating a value for the `positive` parameter with `confusionMatrix(df$predict_type, df$actual_type, positive = "spam")`. The output for this function includes a set of performance measures.

The kappa statistic

The kappa statistic adjusts accuracy by accounting for the possibility of a correct prediction by chance alone. We already talked about the class imbalance problem. The kappa statistic will reward the classifier only if it's better than just always guessing the most frequent class. A common subjective interpretation is as follows:

- Poor agreement = less than 0.2
- Fair agreement = 0.2 to 0.4
- Moderate agreement = 0.4 to 0.6
- Good agreement = 0.6 to 0.8
- Very good agreement = 0.8 to 1

Compara el que s'obté a la diagonal amb el que s'hi obtindria si simplement apliquéssim productes tal com explica a la pàgina 324 i 325.

We can also calculate kappa using `vcd::Kappa()` or `irr::kappa2()`. **But not using built-in** `kappa()`, as it calculates something completely different.

Sensitivity and specificity

Finding a useful classifier often involves a balance between predictions that are overly conservative and overly agressive. Sensitivity and specificity are a pair of performance measures that capture this trade-off. Both range from 0 to 1, with values close to 1 being more desirable (but we'll often want to find an appropriate balance between the two). Visualizations can also assist with understanding the trade-off between sensitivity and specificity.

The **sensitivity** of a model (or **true positive rate**) measures the proportion of positive examples that were correctly classified:

$$sensitivity = \frac{TP}{TP + FN}$$

The **specificity** of a model (or **true negative rate**) measures the proportion of negative examples that were correctly classified:

$$specificity = \frac{TN}{TN + FP}$$

We can also calculate sensitivity and specificity using specific `caret` functions such as:

- `sensitivity(df$predict_type, df$actual_type, positive = "spam")`
- `specificity(df$predict_type, df$actual_type, negative = "ham")`

Precision and recall

These statistics are intended to provide an indication of how interesting and relevant a model's results are, or whether the predictions are diluted by meaningless noise. It is difficult to build a model with both high precision and high recall. It is therefore important to test a variety of models in order to find the combination of precision and recall that will meet the needs of our project.

The **precison** (or **positive predictive value**) is defined as the proportion of positive examples that are truly positive: how often is the model correct when it predicts the positive class? A precise model will only predict the positve class in cases that are very likely to be positive. It will be very trustworthy.

$$precision = \frac{TP}{TP + FP}$$

The **recall** is a mesure of how complete the results are, defined as the number of true positives over the total number of positives. It's the same as sensitivity, but with a slightly different interpretation.

$$recall = \frac{TP}{TP + FN} = sensitivity$$

We can also calculate precision and recall using specific `caret` functions such as:

- `posPredValue(df$predict_type, df$actual_type, positive = "spam")`
- `sensitivity(df$predict_type, df$actual_type, positive = "spam")`

The F-measure

The **F-measure** (or $\mathbf{F_1}$ **score** or **F-score**) combines precision and recall into a single measure using the harmonic mean (since both precision and recall are expressed as proportions between zero and one).

$$F\,measure = \frac{2 \times precision \times recall}{recall + precision} = \frac{2 \times TP}{2 \times TP + FP + FN}$$

As the F-measure describes the model performance in a single number, it provides a convenient way to compare several models side by side assuming equal weight for precision and recall.

Visualizing performance trade-offs

Visualizations depict how a learner performs across a wide range of conditions. It is possible that two models with similar accuracy could have drastic differences in how they achieve their accuracy. Some models may struggle with certain predictions that others make with ease, while breezing through the cases that others cannot get right. Visualizations let us understand these differences.

We'll use the `ROCR` package. And we'll need two vectors of data: one with the predicted class values and another with the estimated probability of the positive class. Every classifier evaluation using `ROCR` starts with creating a `prediction` object as in:

```
pred <- prediction(predictions = df$prob_positive, labels = df$actual_type)
```

And from here on we can use the `performance()` function to compute measures of performance from that object, which we can then plot using R's built-in `plot()` function.

ROC curves

The **Receiver Operating Characteristic (ROC) curve** is commonly used to examine the trade-off between the detection of true positives while avoiding the false positives. In the plot, the vertical axis represents the proportion of true positives (sensitivity), while the horizontal axis represents the

proportion of false positives (1 - specificity). The diagram is also known as a sensitivity/specificity plot.

The points comprising ROC curves indicate the true positive rate at varying false positive thresholds. A test classifier is better if its curve falls closer to something like ↱ (perfect classifier, it correctly identifies all of the positives before it incorrectly classifies any negative result) than to something like ↗ (classifier with no predictive value, it detects true positives and false positives at exactly the same rate).

So, the closer the curve is to the perfect classifier, the better it is at identifying positive values. This can be measured using a statistic known as the **area under the ROC curve (AUC)**. AUC ranges from 0.5 (for a classifier with no predictive value) to 1.0 (for a perfect classifier). A common subjective convention is as follows:

- **A**: Outstanding = 0.9 to 1.0
- **B**: Excellent/good = 0.8 to 0.9
- **C**: Acceptable/fair = 0.7 to 0.8
- **D**: Poor = 0.6 to 0.7
- **E**: No discrimination = 0.5 to 0.6

We must keep in mind two ROC curves may be shaped very differently and yet have identical AUC values. So, qualitative examination of the ROC curve is a must.

As stated before, we can use the `performance()` function to compute measures of performance from the `prediction` object, which we can then plot using R's built-in `plot()` function:

```
perf <- performance(pred, measure = "tpr", x.measure = "fpr")
plot(perf, col = "blue", lwd = 3)
abline(a = 0, b = 1, lwd = 2, lty = 2)
```

And we can use `performance()` again to calculate the AUC:

```
perf.auc <- performance(pred, measure = "auc")
str(perf.auc)
unlist(perf.auc@y.values)
```

When we get a high value for AUC, how do we know wheter the model is just as likely to perform well for another dataset? We need to have a better understanding of how far we can extrapolate a model's predictions beyond the test data.

Estimating future performance

Resubstitution error for a model occurs when the training data is incorrectly predicted in spite of the model being built directly from this data. This information can be used as a rough diagnostic to identify obviously poor performers. But the error rate on the training data can be extremely optimistic about a model's future performance (if the model fits too well to the training data, aka *overfitting*).

Instead of relying on resubstitution error, a better practice is to evaluate a model's performance on data it has not yet seen, for instance splitting the available data into training and test sets. However, this is not always ideal, for instance if we have a small pool of data and don't want to reduce the sample any further. In these scenarios we can use different ways to estimate a model's performance on unseen data.

The holdout method

This is just the procedure of partitioning data into **training and test datasets**. The training dataset is used to generate the model, which is then applied to the test dataset to generate predictions for evaluation. Usually, we use about 2/3 of the data for training and 1/3 for testing, but this can vary depending on the amount of available data.

At no time the performance on the test dataset should be allowed to influence the model. If we build several models on the training data and select the one with the highest accuracy on the test data, the test performance won't be an unbiased measure of the performance on unseen data.

To avoid this problem, it's better to create a third dataset: the **validation dataset**. This dataset would be used for iterating and refining the model or models chosen, leaving the test dataset to be used only once as a final step to report an estimated error rate for future predictions. A 50-25-25 split is pretty typical.

One problem with this method is that in cases in which a class is a very small proportion of the dataset, this can lead this class to be omitted from the training dataset, which will cause the model not being able to learn about it. To help with this we can use **stratified random sampling**, which guarantees that the random partitions will have nearly the same proportion of each class as the full dataset (but does not guarantee other type of representativeness).

A technique called **repeated holdout** can be used to mitigate the problems of randomly composed training datasets. It uses the average result from several random holdout samples to evaluate a model's performance.

Useful R statements:

- `set.seed(42)` ensures that the results are consistent if the same code is run multiple times.
- `random_ids <- order(runif(1000))` generates a random sequence of numbers between 1 and 1000 that can be used to divide a data frame into different datasets.
- `in_train <- createDataPartition(df$some_class, p = 0.75, list = FALSE)` creates a vector than we can then use to create stratified random samplings as easily as `data_train <- df[in_train, ]` and `data_test <- df[-in_train, ]`.

Since models trained on larger datasets generally perform better, a common practice is to retrain the model on the full set of data (training plus test and validation) after a final model has been selected and evaluated.

Cross-validation

The technique known as **k-fold cross-validation** is the industry standard for estimating model performance. It randomly divides the data into **k folds** to completely separate random partitions. The most common convention is to use **10-fold cross-validation**: each of the 10 folds comprises 10% of the total data. A machine learning model is built on 90% of data and evaluated using the other 10%. Process is repeated 10 times (using a different testing fold each time) and the average performance across all the folds is reported.

Datasets for cross-validation can be created with `folds <- createFolds(df$some_class, k = 10)`, which will attempt to maintain the same class balance in each of the folds as in the original dataset. We then use this vector with `data_test <- df[folds$Fold01, ]` and `data_train <- df[-folds$Fold01, ]`.

This process needs to be (programmatically) repeated a total of 10 times, building a model and calculating its performance each time, so that we can average all of them to obtain the overall performance.

A technique called **repeated k-fold CV** (repeatedly applying k-fold CV and averaging the results) is quite computationally intensive, but provides a very robust estimate. A common strategy is to perform 10-fold CV ten times.

Bootstrap sampling

Bootstrapping implies the creation of several randomly selected training and test datasets. The difference with k-fold cross-validation is that bootstrap sampling allows examples to be selected multiple times through a process of sampling with replacement. This means that from the original dataset of n examples we can create one or more new training datasets that will also contain n examples (some of which, of course, are repeated). However, bootstrap samples are less representative of the full dataset than 10-folds. In consequence, this technique's performance estimates may be substantially lower than what would be obtained when the model is later trained on the full dataset.

One advantage of bootstrap over cross-validation is that it tends to work better with very small datasets. Additionally, bootstrap sampling has applications beyond performance measurement (i.e. it can be used to improve model performance).

References

Lantz, Brett. 2015. *Machine Learning with R*. Packt Publishing Ltd.