Overview

After reading this chapter, you will know several techniques for analyzing a model test set to derive insights on likely model performance in the future. These techniques include the same model performance metrics we’ve already calculated, such as the ROC AUC, as well as new kinds of visualizations such as the sloping of default risk by bins of predicted probability, and the calibration of predicted probability. To bridge the gap between the theoretical metrics of machine learning and the financial metrics of the business world, you will also learn what to consider when estimating the financial impact of a model and providing guidance to the client on how to realize this impact. We close with a discussion of practical concerns of delivering and deploying a model, such as the format of delivery and ways to monitor the model as it is being used.

Introduction

In the previous chapter, we saw how XGBoost could be used to train a model with better performance than our previous efforts, and how to explain models using SHAP values. In this chapter, we will consider model building to be complete and address the remaining issues that need attention before delivering the model to the client. The key elements of this chapter are analysis of the test set, including financial analysis, and issues to consider when delivering a model to a client who wants to use it in the real world.

We look at the test set to get an idea of how well the model will perform in the future. By calculating metrics we already know, like ROC AUC, but now on the test set, we can gain confidence that our model will be useful for new data. We’ll also learn some intuitive ways to visualize the power of the model for grouping customers into different levels of risk of default, such as a decile chart.

Your client will likely appreciate the efforts you made in creating a more accurate model or one with higher ROC AUC. However, they will definitely appreciate understanding how much money the model can help them earn or save and will probably be happy to receive specific guidance on how to maximize the model's potential for this. A financial analysis on the test set can simulate different scenarios of model-based strategies and help the client pick one that works for them.

After completing the financial analysis, we wrap up by discussing issues of how to deliver a model for use by the client and how to monitor its performance over time.

Review of Modeling Results

In order to develop a binary classification model to meet the business requirements of our client, we have now tried several modeling techniques with varying degrees of success. In the end, we’d like to choose the model with the best performance, to do further analyses on and present to our client. However, it is also good to communicate the other options we explored, demonstrating a thoroughly researched project.

Here, we review the different models that we tried for the case study problem, the hyperparameters that we needed to tune, and results from cross-validation, or the validation set in the case of XGBoost. We only include the work we did using all possible features, not the earlier models where we used only one or two features.

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Location in book | Tuned hyperparameters | Validation ROC AUC |
| Logistic regression with L1 regularization | Chapter 4 Activity | regularization parameter *C* | 0.719 |
| Logistic regression with L1 regularization and interaction features | Chapter 4 Activity | regularization parameter *C* | 0.739 |
| Decision tree | Chapter 5 Exercise 2 | Maximum depth | 0.746 |
| Random forest | Chapter 5 Activity | Maximum depth and number of trees | 0.776 |
| XGBoost | Chapter 6 Activity | Maximum leaves | 0.779 |

Figure 7.1: Summary of modeling activities with case study data

When presenting results to the client, you should be prepared to interpret them for business partners at all levels of technical familiarity, including those with very little technical background. For example, business partners may not understand the derivation of the ROC AUC measure, however this is an important concept since it’s the main performance metric we used to assess models. You may need to explain that it's a metric that can vary between 0.5 and 1 and give intuitive explanations for these limits: 0.5 is no better than a coin flip and 1 is perfection, which is essentially unattainable.

Our results are somewhere in between, getting close to 0.78 with the best model we developed. While the ROC AUC of a given model may not necessarily be meaningful by itself, Figure 7.1 shows that we've tried several methods and have achieved improved performance above our initial attempts. In the end, for a business application like the case study, abstract model performance metrics like ROC AUC should be accompanied by a financial analysis if possible. We will explore this later in this chapter.

Note on Interpreting the ROC AUC

An interesting interpretation of the ROC AUC score is the probability that for two samples, one with a positive outcome and one with a negative outcome, the positive sample will have a higher predicted probability than the negative sample. In other words, for all possible pairs of positive and negative samples in the dataset being assessed, the proportion of pairs where the positive sample has a higher model prediction than the negative sample is equivalent to the ROC AUC.

From Figure 7.1, we can see that for the case study, our efforts in creating more complex models, either by engineering new features to add to a simple logistic regression, or by creating an ensemble of decision trees, yielded better model performance. In particular the random forest and XGBoost models perform similarly, although these validation scores are technically not directly comparable since in the case of random forest we excluded missing values and used 4-fold cross-validation, while for XGBoost the missing values were included and there was just one validation set which was used for early stopping. However, Figure 7.1 provides an indication that either XGBoost or random forest would probably be the best choices. We’ll move forward here with the XGBoost model.

Now that we’ve decided which model we’ll deliver, it’s good to consider additional things we could have tried in the model development process. These concepts won’t be explored in this book, but you may wish to experiment with them on your own.

Feature Engineering

Another way to increase model performance that we touched on briefly is feature engineering. While we used scikit-learn’s automated feature engineering capabilities to make interaction features, you can also manually engineer features from existing features. For example, a credit account that is using a large percentage of its credit limit may be considered particularly risky. We have information in our features about the credit limit, and also the amounts of past bills, which we could use to engineer a new feature: the ratio of past monthly billed amounts to the credit limit, assuming the billed amount is the account’s balance. This measure of credit utilization may be a stronger feature and result in better model performance when calculated in this way, than having the credit limit and monthly billed amounts available to the model separately.

Feature engineering may take the form of manipulating existing features to make new ones, as in the previous example, or it may involve bringing in entirely new data sources and creating features with them.

Ensembling Multiple Models

In choosing the final model to deliver for the case study project, it would probably be fine to deliver either the random forest or the XGBoost. Another commonly used approach in machine learning is to ensemble together multiple models. This means combining the predictions of different models, similar to how random forest and XGBoost combine many decision trees. But in this case the way to combine model predictions is up to the data scientist. A simple way to create an ensemble of models is to take the average of their predictions.

Ensembling is often done when there are multiple models, perhaps different kinds of models or models trained with different features, that all have good performance. In our case, it may be that using the average prediction from the random forest and the XGBoost would have better performance than either model on its own. To explore this, we could compare performance on a validation set, for example the one used for early stopping in XGBoost.

Different Modeling Techniques

Depending on how much time you have for a project and your expertise in different modeling techniques, you would want to try as many methods as possible. More advanced methods, such as neural networks for classification, may yield improved performance on this problem. We encourage you to continue your studies and learn how to use these models. However, for tabular data such as what we have for the case study, XGBoost is a good de facto choice and will likely provide excellent, if not the best performance of all methods.

Balancing Classes

Note that we did not address the class imbalance in the response variable. You are encouraged to try fitting models with the class\_weight='balanced' option in scikit-learn or using the scale\_pos\_weight hyperparameter in XGBoost, to see the effect.

While these would be interesting avenues for further model development, for the purposes of this book, we are done with model building at this point. We’ll move forward to examine XGBoost model performance on the test set.

Model Performance on the Test Set

We already have some idea of the out-of-sample performance of the XGBoost model, from the validation set. However, the validation set was used in model fitting, via early stopping. The most rigorous estimate of expected future performance we can make should be created with data that was not used at all for model fitting. This was the reason for reserving a test dataset from the model building process.

You may notice that we did examine the test set to some extent already, for example in the first chapter when assessing data quality and doing data cleaning. The gold standard for predictive modeling is to set aside a test set at the very beginning of a project and not examine it at all until the model is finished. This is the easiest way to make sure that none of the knowledge from the test set has “leaked” into the training set during model development. When this happens, it opens up the possibility that the test set is no longer a realistic representation of future, unknown data. However, it is sometimes convenient to explore and clean all of the data together, as we’ve done. If the test data has the same quality issues as the rest of the data, then there would be no leakage. It is most important to make sure you’re not looking at the test set when you decide which features to use, fit various models, and compare their performance.

We begin the test set examination by loading the trained model from the Chapter 6 Activity, along with the training and test data and feature names, using Python’s pickle:

with open('../../Data/xgb\_model\_w\_data.pkl', 'rb') as f:

features\_response, X\_train\_all, y\_train\_all, X\_test\_all,\

y\_test\_all, xgb\_model\_4 = pickle.load(f)

With these variables loaded in the notebook, we can make predictions for the test set and analyze them. First obtain the predicted probabilities for the test set:

test\_set\_pred\_proba = xgb\_model\_4.predict\_proba(X\_test\_all)[:,1]

Now import the ROC AUC calculation routine from scikit-learn, use it to calculate this metric for the test set, and display it:

from sklearn.metrics import roc\_auc\_score

test\_auc = roc\_auc\_score(y\_test\_all, test\_set\_pred\_proba)

test\_auc

The result should be as follows:

0.7735528979671706

The ROC AUC of 0.774 on the test set is a bit lower than the 0.779 we saw on the validation set for the XGBoost model, however it is not very different. Since the model fitting process optimized the model for performance on the validation set, it’s not totally surprising to see somewhat lower performance on new data. Overall, the testing performance is in line with expectations and we can consider this model successfully tested in terms of the ROC AUC metric.

While we won’t do this here, a final step before delivering a trained model might be to fit it on all of the available data, including the unseen test set. This could be done by concatenating the training and testing data features (X\_train\_all, X\_test\_all) and labels (y\_train\_all, y\_test\_all), and using them to fit a new model, perhaps by defining a new validation set for early stopping or using the current test set for that purpose. This approach is motivated by the idea that machine learning models generally perform better when trained on more data. The downside is that since there would be no unseen test set in these circumstances, the final model could be considered to be untested.

Data scientists have varying opinions on which approach to use: only using the unseen test set for model assessment, versus using as much data as possible, including the test set, to train the final model once all previous steps in the process are completed. One consideration is whether or not a model would benefit from being trained on more data. This could be determined by constructing a learning curve. Although we won’t illustrate this here, the concept behind a learning curve is to train a model on successively increasing amounts of data and calculating the validation score on the same validation set. For example, if you had 10,000 training samples, you might set aside 500 as a validation set and then train a model on the first 1,000 samples, then the first 2,000 samples, and so on up to all 9,500 samples that aren’t in the validation set. If training on more data consistently increases the validation score even up to the point of using all available data, this is a signal that training on more data than you have in the training set would be beneficial. However, if model performance starts to level off at some point and it doesn’t seem like additional data would create a more performant model, you may not need to do this. Learning curves can provide guidance on which approach to take with the test set, as well as whether more data is needed in a project generally.

For the purposes of the case study, we’ll assume that we wouldn’t realize any benefit from refitting the model using the test set. So, our main concerns now are presenting the model to the client, helping them design a strategy to use it to meet their business goals, and providing guidance on how the model's performance can be monitored as time goes on.

Distribution of Predicted Probability and Decile Chart

The ROC AUC metric is good because it provides a single number which summarizes model performance on a dataset. However, it’s also insightful to look at model performance for different subsets of the population. One way to break up the population into subsets is to use the model predictions themselves. Using the test set, we can visualize the predicted probabilities with a histogram:

mpl.rcParams['figure.dpi'] = 400

plt.hist(test\_set\_pred\_proba, bins=50)

plt.xlabel('Predicted probability')

plt.ylabel('Number of samples')

This code should produce the following plot:

Chart, histogram

Description automatically generated

Figure 7.2: Distribution of predicted probabilities for test set

The histogram of predicted probabilities for the test set shows that most predictions are clustered in the range [0, 0.2]. In other words, most borrowers have between a 0 and 20% chance of default, according to the model. However there appears to be small cluster of borrowers with higher risk, centered near 0.7.

As a visually intuitive way to examine model performance for different regions of predicted default risk is to create a decile chart, which groups borrowers together based on the decile of predicted probability. Within each decile, we can compute the true default rate. We would expect to see a steady increase in default rate from the lowest prediction deciles to the highest.

We can compute deciles like we did in Exercise 6.01, using pandas’ qcut:

deciles, decile\_bin\_edges = pd.qcut(x=test\_set\_pred\_proba,

q=10,

retbins=True)

Here we are splitting the predicted probabilities for the test set, supplied with the x keyword argument. We want to split them into 10 equal size bins, with the bottom 10% of predicted probabilities in the first bin, and so on, so we indicate we want q=10 quantiles. However, you can split into any number of bins you want, such as 20 (ventiles) or 5 (quintiles). Since we indicate retbins=True, the bin edges are returned in the variable decile\_bin\_edges, while the series of decile labels is in deciles. We can examine the 11 bin edges needed to create 10 bins:

decile\_bin\_edges

which should produce:

array([0.02213463, 0.06000734, 0.08155108, 0.10424594, 0.12708404,

0.15019046, 0.18111563, 0.23032923, 0.32210371, 0.52585585,

0.89491451])

In order to make use of the decile series, we can combine it with the true labels for the test set, and the predicted probabilities, into a dataframe:

test\_set\_df = pd.DataFrame({'Predicted probability':test\_set\_pred\_proba,

'Prediction decile':deciles,

'Outcome':y\_test\_all})

test\_set\_df.head()

The first few rows of the dataframe should look like this:

Table

Description automatically generated

Figure 7.3: Dataframe with predicted probabilities and deciles

In the dataframe we can see that each sample is labelled with a decile bin, indicated using the bin edges that contain the predicted probability. The outcome shows the true label. What we want to show in our decile chart is the true default rate within the decile bins. For this we can use pandas’ groupby capabilities. First we create a groupby object, by grouping our dataframe on the decile column:

test\_set\_gr = test\_set\_df.groupby('Prediction decile')

The groupby object can be aggregated by other columns. In particular here we’re interested in the default rate within decile bins, which is the mean of the outcome variable. We also calculate a count of the data in each bin. Since quantiles, such as deciles, group the population into equal sized bins, we expect the counts to be the same or similar.

gr\_df = test\_set\_gr.agg({'Outcome':['count', 'mean']})

Examine our grouped dataframe gr\_df:

Table

Description automatically generated

Figure 7.4: Default rate in deciles of predicted probability on the test set

In Figure 7.4, we can see that indeed the counts are nearly equal in all bins. We also can tell that the true default rate increases with decile, as we hope and expect since we know our model has good performance. Before visualizing the data, it’s worth noting that this dataframe has a special kind of column index called a multiindex. Notice that there are two lines of text describing the columns, a top-level index that only contains one label Outcome and a 2nd level index with the labels count and mean. Accessing data in dataframes that have a multiindex is a little more complicated than for the dataframes we’ve worked with previously. We can display the column index as follows:

gr\_df.columns

which should produce the result:

MultiIndex([('Outcome', 'count'),

('Outcome', 'mean')],

)

Here we can see that to access a column from a multiindex, we need to use tuples that specify each level of the index, for example gr\_df[('Outcome','count')]. While here the multiindex isn’t really necessary since we’ve only done an aggregation of one column (Outcome), it can come in handy when there are aggregations on multiple columns.

Now we’d like to create a visualization, showing how the model predictions do a good job of binning borrowers into groups with consistently increasing default risk. We’re going to show the counts in each bin, as well as the default risk in each bin. Because these columns are on different scales, with counts in the hundreds and risk between 0 and 1, we should use a dual y-axis plot. In order to have more control over plot appearance, we’ll create this plot using matplotlib functions instead of doing it through pandas. First we create the plot of sample size in each bin, labeling the y-axis ticks with the same color as the plot for clarity. Please see the notebook on github if you’re reading in black and white, as color is important for this plot. This code snippet should be run in the same cell as the next one. Here we create a set of axes, then add a plot to it along with some formatting and annotation:

ax\_1 = plt.axes()

color\_1 = 'tab:blue'

gr\_df[('Outcome', 'count')].plot.bar(ax=ax\_1, color=color\_1)

ax\_1.set\_ylabel('Count of observations', color=color\_1)

ax\_1.tick\_params(axis='y', labelcolor=color\_1)

ax\_1.tick\_params(axis='x', labelrotation = 45)

Notice that we’re creating a bar plot for the sample sizes. We’d like to add a line plot to this, showing the default rate in each bin on a right-hand y-axis, but the same x-axis as the existing plot. matplotlib makes a method called twinx available for this purpose, which can be called on an axes object to return a new axes object sharing the same x-axis. We take similar steps to then plot the default rate and annotate:

ax\_2 = ax\_1.twinx()

color\_2 = 'tab:red'

gr\_df[('Outcome', 'mean')].plot(ax=ax\_2, color=color\_2)

ax\_2.set\_ylabel('Fraction of defaulters', color=color\_2)

ax\_2.tick\_params(axis='y', labelcolor=color\_2)

After running the above two snippets in a code cell, the following plot should appear:

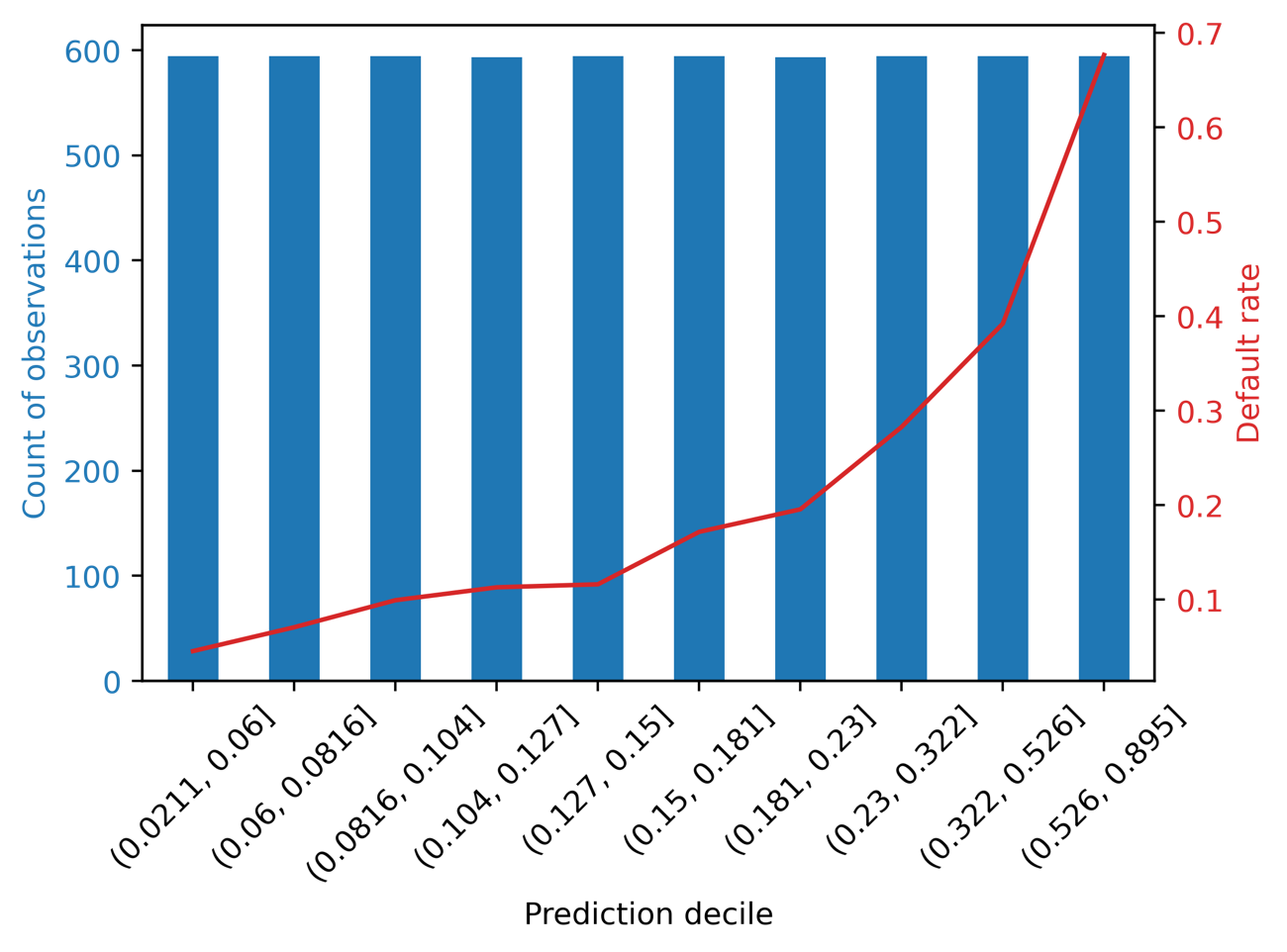


Figure 7.5: Default rate according to model prediction decile

Figure 7.5 contains the same information as displayed in the dataframe in Figure 7.4, but in a nicer presentation. It’s clear that default risk increases with each decile, where the riskiest 10% of borrowers have a default rate close to 70%, but the least risky are below 10%. When a model is able to effectively distinguish groups of borrowers with consistently increasing default risk, the model is said to slope the population being examined. Notice also that the default rate is relatively flat across the lowest 5 to 7 deciles, likely because these observations are mostly clustered in the range [0, 0.2] of predicted risk as seen in the histogram in Figure 7.2.

Splitting the test set into equal-population deciles is one way to examine model performance, in terms of sloping default risk. However, a client may be interested in looking at default rate by different groups, such as equal interval bins (e.g., binning together all observations in the prediction ranges [0, 0.2), [0.2, 0.4) etc. regardless of sample size in each bin), or some other way. You’ll explore how to easily do this in pandas in the following exercise.

Notes on Uncertainty in Default Rate

In the following exercise we’ll make use of a couple statistical concepts to help create error bars, including the standard error of the mean and the normal approximation to the binomial distribution.

The standard error of the mean is also known as the standard deviation of the sampling distribution of the sample mean. That is a long name, but the concept isn’t too complicated. The idea behind this is that the population of samples we have in the bin that we wish to make error bars for represents one possible way of sampling a theoretical larger population of similar samples. If we could take repeated samples from the larger population, each of these sampling events would result in a slightly different mean (the sample mean). Constructing a distribution of these means (the sampling distribution of the sample mean) from repeated sampling would allow us to know the variance of this sampling distribution, which would be useful as a measure of uncertainty in the mean that we’ve measured from our data. It turns out this variance (let’s call it ) depends on the sample size of the population we observe (*n*): it is inversely proportional to sample size, but also directly proportional to the variance of the larger, unobserved population (*σ*): .

While we don’t know *σ*, it can be estimated by the variance in the sample we observed. This observed variance can be calculated as *p*(1-*p*), where *p* is the proportion of successes, or defaults for the case study. Given the formula for the variance of the sampling distribution of the sample mean above, we can plug in the observed variance, then take the square root to get the standard error of the mean: . This formula is also known as the normal approximation to the binomial distribution in some contexts. We’ll use it below to create error bars on an equal-interval chart of default rate for different model prediction bins. For more details on these concepts, the reader is encouraged to consult a statistics textbook.

Exercise 7.01: Equal-interval Chart

In this exercise you will make a similar chart to that shown in Figure 7.5, however instead of splitting the test set into equal-population deciles of predicted probability, you’ll use equal intervals of predicted probability. Specifying the intervals could be helpful if a business partner wants to think about potential model-based strategies using certain score ranges. You can use pandas cut to create equal-interval binnings, or custom binnings using an array of bin edges, similar to how you used qcut to create quantile labels.

1. Create the series of equal-interval labels, for 5 bins, using the following code:

equal\_intervals, equal\_interval\_bin\_edges = \

pd.cut(x=test\_set\_pred\_proba,

bins=5,

retbins=True)

Notice this is similar to the call to qcut, except here with cut we can say how many equal-interval bins we want by supplying an integer to the bins argument. You could also supply an array for this argument to specify the bin edges, for custom bins.

1. Examine the equal-interval bin edges with this code:

equal\_interval\_bin\_edges

The result should be:

array([0.02126185, 0.1966906 , 0.37124658, 0.54580256, 0.72035853,

0.89491451])

You can confirm that these bin edges have equal intervals between them by subtracting the subarray going from the 1st to the next-to-last item, from the subarray starting with the 2nd and going to the end.

1. Check the intervals between bin edges like this:

equal\_interval\_bin\_edges[1:] - equal\_interval\_bin\_edges[:-1]

The result should be:

array([0.17542876, 0.17455598, 0.17455598, 0.17455598, 0.17455598])

You can see that the distance between the bin edges is roughly equal. The first bin edge is a bit smaller than the minimum predicted probability, as you can confirm for yourself.

In order to create a similar plot to Figure 7.5, first we need to put the bin labels together with the response variable in a dataframe, as we did previously with decile labels. We also put the predicted probabilities in the dataframe for reference.

1. Make a dataframe of predicted probabilities, bin labels, and the response variable for the test set like this:

test\_set\_bins\_df = pd.DataFrame({'Predicted probability':test\_set\_pred\_proba,

'Prediction decile':equal\_intervals,

'Outcome':y\_test\_all})

test\_set\_bins\_df.head()

The result should look as follows:

Table

Description automatically generated

Figure 7.6: Dataframe with equal-interval bins

We can use this dataframe to group by the bin labels, then get the metrics we are interested in: aggregations that represent the default rate and the number of samples in each bin.

1. Group by the bin label and calculate the default rate and sample count within bins with this code:

test\_set\_equal\_gr = test\_set\_bins\_df.groupby('Prediction decile')

gr\_eq\_df = test\_set\_equal\_gr.agg({'Outcome':['count', 'mean']})

gr\_eq\_df

The resulting dataframe should appear like this:

Table

Description automatically generated

Figure 7.7: Grouped data for 5 equal-interval bins

Notice that here, unlike with quantiles, there are a different number of samples in each bin. The default rate appears to increase across bins in a consistent manner. Let’s plot this dataframe to create a similar visualization to Figure 7.5.

Before creating this visualization, in order to consider that the estimates of default rate may be less robust for higher predicted probabilities, due to decreased sample size in these ranges, we’ll calculate the standard error of the default rates.

1. Calculate the standard errors of default rates within bins using this code.

p = gr\_eq\_df[('Outcome', 'mean')].values

n = gr\_eq\_df[('Outcome', 'count')].values

std\_err = np.sqrt(p \* (1-p) / n)

std\_err

The result should appear as follows:

array([0.00506582, 0.01258848, 0.02528987, 0.02762643, 0.02683029])

Notice that for the bins with higher score ranges and fewer samples, the standard error is larger. It will be helpful to visualize these standard errors with the default rates.

1. Use this code to create an equal-interval plot of default rate and sample size. The code is very similar to that needed for Figure 7.5, except here we include error bars on the default rate plot using the yerr keyword and the results from the previous step:

ax\_1 = plt.axes()

color\_1 = 'tab:blue'

gr\_eq\_df[('Outcome', 'count')].plot.bar(ax=ax\_1, color=color\_1)

ax\_1.set\_ylabel('Count of observations', color=color\_1)

ax\_1.tick\_params(axis='y', labelcolor=color\_1)

ax\_1.tick\_params(axis='x', labelrotation = 45)

ax\_2 = ax\_1.twinx()

color\_2 = 'tab:red'

gr\_eq\_df[('Outcome', 'mean')].plot(ax=ax\_2, color=color\_2,

yerr=std\_err)

ax\_2.set\_ylabel('Default rate', color=color\_2)

ax\_2.tick\_params(axis='y', labelcolor=color\_2)

The result should appear like this:

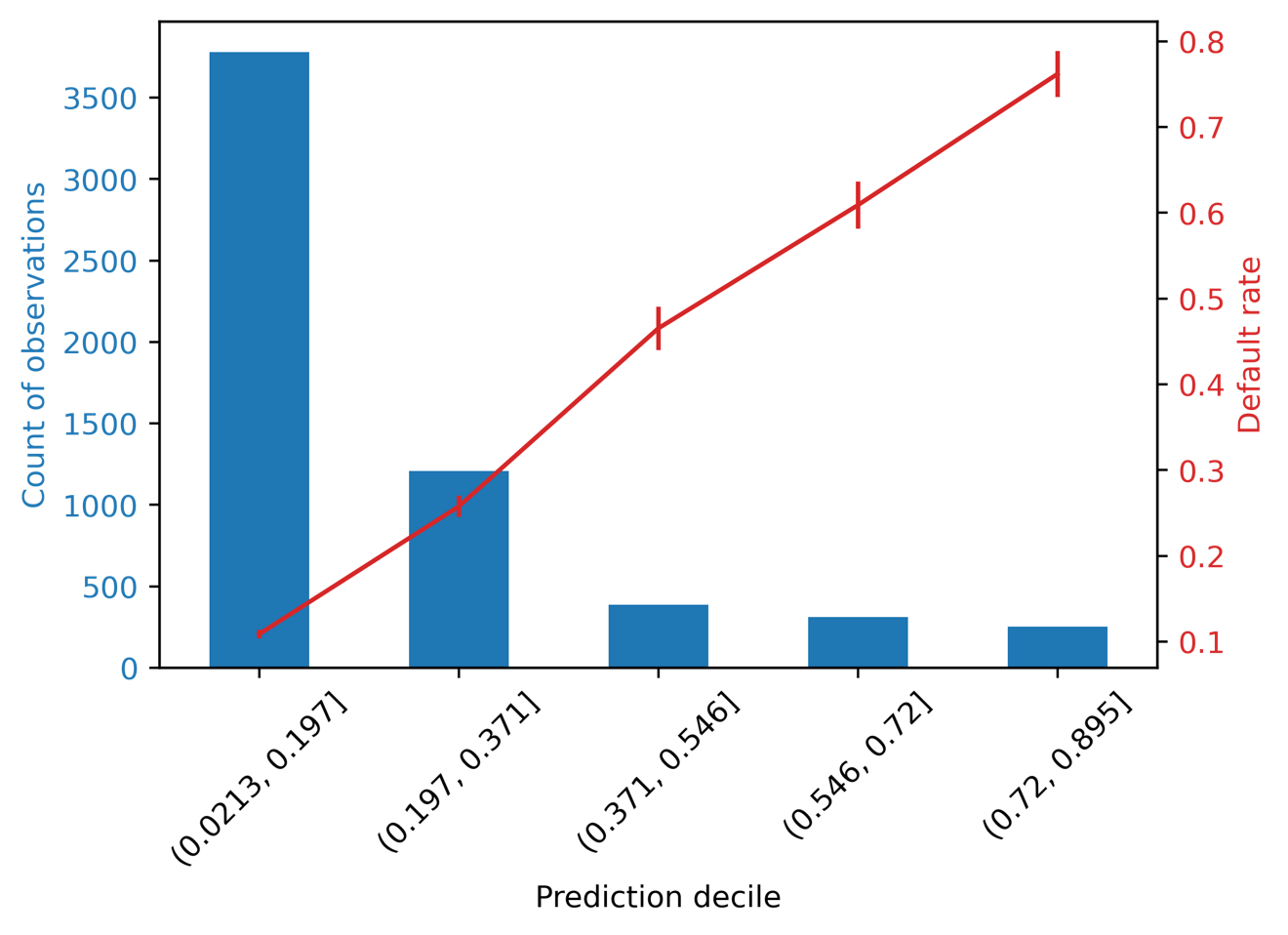


Figure 7.8: Plot of default rate and sample count for equal-interval bins

We can see in Figure 7.8 that the number of samples is pretty different among the different bins, in contrast to the quantile approach. While there are relatively few samples in the higher score bins, the error bars on the plot of default rate are small compared to the overall trend of increasing default rate from lower to higher score bins, so we can be confident in this trend.

Calibration of Predicted Probabilities

One interesting feature of Figure 7.8 is that the line plot of default rates increases by roughly the same amount from bin to bin. Contrast this to the decile plot in Figure 7.5, where the default rate increases slowly at first and then more rapidly. Notice also that the default rate for each bin appears to be roughly the midpoint of each bin. This implies that the default rate is similar to the average model prediction in each bin. In other words, not only does our model appear to effectively rank borrowers from low to high risk of default, as quantified by the ROC AUC, it also appears to accurately predict the probability of default.

Measuring how closely predicted probabilities match actual probabilities is the goal of calibrating probabilities. A standard measure for probability calibration follows from the concepts discussed above and is called expected calibration error (ECE), defined as

where index *i* ranges from 1 to the number of bins (*N*), *F(i)* is the fraction of all samples falling in bin *i*, *oi* is the fraction of samples in bin *i* that are positive (i.e., for the case study, defaulters), and *ei* is the average of predicted probabilities within bin *i*.

We can calculate the ECE for the predicted probabilities within decile bins of the test set using a dataframe very similar to that shown in Figure 7.4, needed to create the decile chart. The only addition we need is the mean predicted probability in each bin. Create such a dataframe as follows:

cal\_df = test\_set\_gr.agg({'Outcome':['count', 'mean'],

'Predicted probability':'mean'})

cal\_df

The output dataframe should look like this:

Table

Description automatically generated

Figure 7.9: Dataframe for calculating ECE metric

For convenience let’s define a variable for F, which is the fraction of samples in each bin. This is the counts in each bin from the above dataframe divided by the total number of samples, taken from the shape of the response variable for the test set:

F = cal\_df[('Outcome', 'count')].values/y\_test\_all.shape[0]

F

The output should be

array([0.10003368, 0.10003368, 0.10003368, 0.09986527, 0.10003368,

0.10003368, 0.09986527, 0.10003368, 0.10003368, 0.10003368])

So, each bin has about 10% of the samples. This is expected, of course, since the bins were created using a quantile approach. However for other binnings, the sample sizes in the bins may not be equal. Now let’s implement the formula for ECE in code, to calculate this metric:

ECE = np.sum(

F

\* np.abs(

cal\_df[('Outcome', 'mean')]

- cal\_df[('Predicted probability', 'mean')]))

ECE

The output should be

0.008144502190176022

This number represents the ECE for our final model, on the test set. By itself, the number isn’t all that meaningful, however metrics like this can be monitored over time, after the model has been put in production and is being used in the real world. If the ECE starts to increase, this is a signal that the model is becoming less calibrated and may need to be retrained, for example, or have a calibration procedure applied to the outputs.

A more intuitive way to examine the calibration of our predicted probabilities for the test set is to plot the ingredients needed for ECE, in particular the true default rate of the response variable, against the average of model predictions in each bin. To this we add a 1-1 line, that represents perfect calibration, as a point of reference:

ax = plt.axes()

ax.plot([0, 0.8], [0, 0.8], 'k--', linewidth=1,

label='Perfect calibration')

ax.plot(cal\_df[('Outcome', 'mean')],

cal\_df[('Predicted probability', 'mean')],

marker='x',

label='Model calibration on test set')

ax.set\_xlabel('True default rate in bin')

ax.set\_xlabel('Average model prediction in bin')

ax.legend()

The resulting plot should look like this:

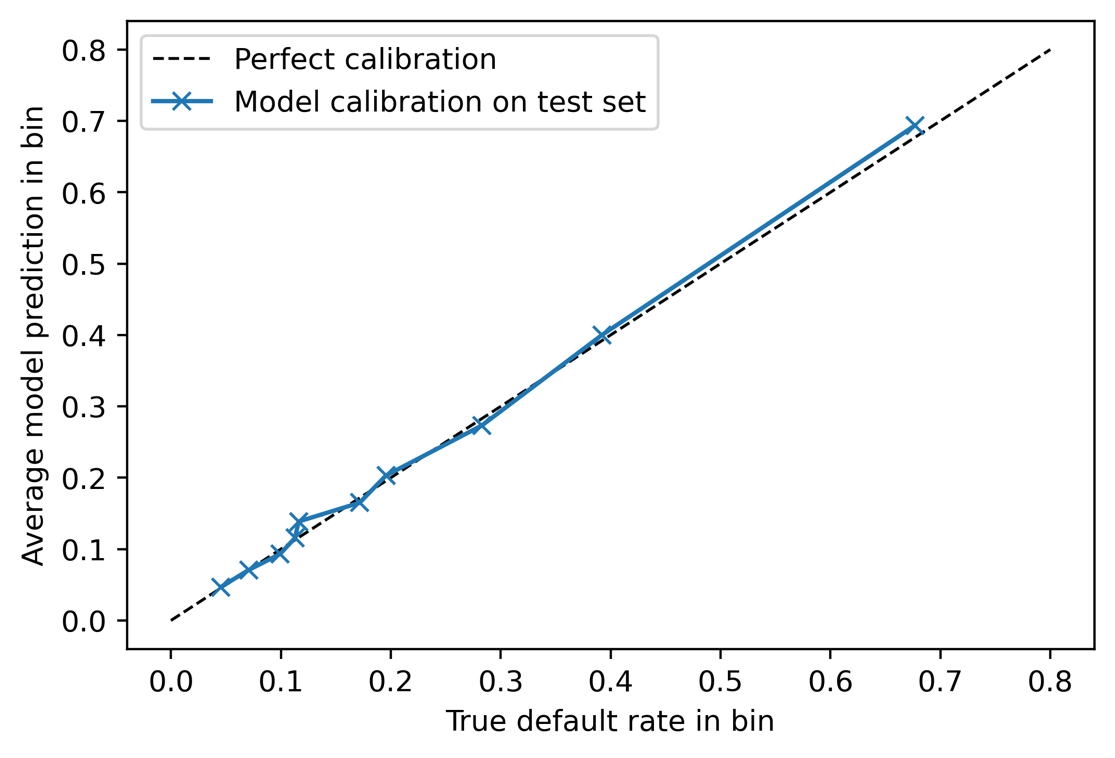


Figure 7.10: Calibration plot for predicted probabilities

Figure 7.10 shows that model-predicted probabilities are very close to the true default rates, so the model appears to be well calibrated. For additional insight, you can try adding error bars to this plot yourself as an exercise. Also note that scikit-learn makes a function available to calculate the information needed to create Figure 7.10: sklearn.calibration.calibration\_curve. However this function does not return the sample size in each bin.

One additional point to be aware of for probability calibration is that some methods for dealing with class imbalance, such as oversampling or undersampling, change the class fraction in the training data set, which will affect the predicted probabilities and likely make them less accurate. This may not be that important though, compared to the ability of the model to rank borrowers on their risk of default, as measured by the ROC AUC, depending on the needs of the client.

Financial Analysis

The model performance metrics we have calculated so far were based on abstract measures that could be applied to analyze any classification model: how accurate a model is, how skillful a model is at identifying true positives relative to false positives at different thresholds (ROC AUC), the correctness of positive predictions (precision), or intuitive measures like sloping risk. These metrics are important for understanding the basic workings of a model and are widely used within the machine learning community, so it's important to understand them. However, for the application of a model to business use cases, we can’t always directly use such performance metrics to create a strategy for how to use the model to guide business decisions, or figure out how much value a model is expected to create. To go the extra mile and connect the mathematical world of predicted probabilities and thresholds, to the business world of costs and benefits, a financial analysis of some kind is usually needed.

In order to help the client with this analysis, the data scientist needs to understand what kinds of decisions and actions might be taken, based on predictions made by the model. This should be the topic of a conversation with the client, preferably early on in the project lifecycle. We have left it until the end of the book so that we could establish a baseline understanding of what predictive modeling is and how it works. However, learning the business context around model usage at the beginning of a project allows you to set goals for model performance in terms of the creation of value, that you can track throughout a project like we tracked the ROC AUC of different models we built. Translating model performance metrics into financial terms is the topic of this section.

For a binary classification model like that of the case study, here are a few questions that the data scientist needs to know the answers to, in order to help the client figure out how to use the model:

* What kinds of decisions does the client want to use the model to help them make?
* How can the predicted probabilities of a binary classification model be used to help make these decisions?
* Are they yes/no decisions? Then choosing a single threshold of predicted probability will be sufficient.
* Are there more than two levels of activity that will be decided on, based on model results? Then choosing 2 or more thresholds, to sort predictions in to low, medium, and high risk for example, may be the solution. For instance, predicted probabilities below 0.5 may be considered low risk, those between 0.5 and 0.75 medium risk, and those above 0.75 high risk.
* What are the costs of taking all the different courses of action that are available, based on model guidance?
* What are the potential benefits to be gained from successful actions taken as a result of model guidance?

Financial Conversation with the Client

We ask the case study client about the points outlined above and learn the following: for credit accounts that are at a high risk of default, the client is designing a new program to provide individualized counseling for the account holder, to encourage them to pay their bill on time or provide alternative payment options if that will not be possible. Credit counseling is performed by trained customer service representatives who work in a call center. The cost per counseling session is NT$7,500 and the expected success rate of a session is 70%, meaning that on average 70% of the recipients of phone calls offering counseling will pay their bill on time, or make alternative arrangements that are acceptable to the creditor. The potential benefits of successful counseling are that the amount of an account's monthly bill will be realized as savings, if it was going to default but instead didn't, as a result of the counseling. Currently, the monthly bills for accounts that default are reported as losses.

After having the above conversation with the client, we have the materials we need to make a financial analysis. The client would like us to help them decide which members to contact and offer credit counseling to. If we can help them narrow down the list of people who will be contacted for counseling, we can help save them money by avoiding unnecessary and expensive contacts. The clients' limited resources for counseling will be more appropriately spent on accounts that are at higher risk of default. This should create greater savings due to prevented defaults. Additionally, the client lets us know that our analysis can help them request a budget for the counseling program, if we can give them an idea of how many counseling sessions it would be worthwhile to offer.

As we proceed to the financial analysis, we see that the decision that the model will help the client make, on an account by account basis, is a yes/no decision: whether to offer counseling to the holder of a given account. Therefore, our analysis should focus on finding an appropriate threshold of predicted probability, by which we may divide our accounts in to two groups: higher risk accounts that will receive counseling and lower risk ones that won't.

Exercise 7.02: Characterizing Costs and Savings

The connection between model output and business decisions the client will make comes down to selecting a threshold for the predicted probabilities. Therefore, in this exercise, we will characterize the expected costs of the counseling program, in terms of costs of offering individual counseling sessions, as well as the expected savings, in terms of prevented defaults, at a range of thresholds. There will be a different cost and savings at each threshold, because each threshold is expected to result in a different number of positive predictions, as well as a different number of true positives within these. The first step is to create an array of potential thresholds. We will use 0 through 1, going by an increment of 0.01. Perform the following steps to complete the exercise.

Note

The code and resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <add link>. Additional steps to prepare data for this exercise, based on previous results in this chapter, have been added to the notebook.

1. Create a range of thresholds to calculate expected costs and benefits of counseling with this code:

thresholds = np.linspace(0, 1, 101)

This creates 101 linearly spaced points between 0 and 1, inclusive.

Now, we need to know the potential savings of a prevented default. To calculate this precisely, we would need to know the next month's monthly bill. However, the client has informed us that this will not be available at the time they need to create the list of account holders to be contacted. Therefore, in order to estimate the potential savings, we will use the most recent monthly bill.

We will use the testing data to create this analysis, as this provides a simulation of how the model will be used after we deliver it to the client: on new accounts that weren't used for model training.

1. Confirm the index of the testing data features array that corresponds to the most recent month's bill:

features\_response[5]

The output should be:

'BILL\_AMT1'

The index 5 is for the most recent months' bill, which we’ll use later.

1. Store the cost of counseling in a variable to use for analysis:

cost\_per\_counseling = 7500

We also know from the client that the counseling program isn't 100% effective. We should take this into account in our analysis.

1. Store the effectiveness rate the client gave us for use in analysis:

effectiveness = 0.70

Now, we will calculate costs and savings for each of the thresholds. We'll step through each calculation and explain it, but for now, we need to create empty arrays to hold the results for each threshold.

1. Create empty arrays to store analysis results. We'll explain what each one will hold in the following steps:

n\_pos\_pred = np.empty\_like(thresholds)

total\_cost = np.empty\_like(thresholds)

n\_true\_pos = np.empty\_like(thresholds)

total\_savings = np.empty\_like(thresholds)

These create empty arrays with the same number of elements as there are thresholds in our analysis. We will loop through each threshold value to fill these arrays.

1. Make a counter variable and open a for loop to go through thresholds:

counter = 0

for threshold in thresholds:

For each threshold, there will a different number of positive predictions, according to how many predicted probabilities are above that threshold. These correspond to accounts that are predicted to default. Each account that is predicted to default will receive a counseling phone call, which has a cost associated with it. So, this is the first part of the cost calculation.

1. Determine which accounts get positive predictions, at this threshold:

pos\_pred = test\_set\_pred\_proba > threshold

pos\_pred is a Boolean array. The sum of pos\_pred indicates the number of predicted defaults, at this threshold.

1. Calculate the number of positive predictions for the given threshold:

n\_pos\_pred[counter] = sum(pos\_pred)

1. Calculate the total cost of counseling for the given threshold:

total\_cost[counter] \

= n\_pos\_pred[counter] \* cost\_per\_counseling

Now that we have characterized the possible costs of the counseling program, at each threshold, we need to see what the projected savings are. Savings are obtained when counseling is offered to the right account holders: those who would otherwise default. In terms of the classification problem, these are positive predictions, where the true value of the response variable is also positive. In other words, true positives.

1. Determine which accounts are true positives, based on the array of positive predictions and the response variable:

true\_pos = pos\_pred & y\_test\_all.astype(bool)

1. Calculate the number of true positives as the sum of the true positive array:

n\_true\_pos[counter] = sum(true\_pos)

The savings we can get from successfully counseling account holders who would otherwise default depends on the savings per prevented default, as well as the effectiveness rate of counseling. We won't be able to prevent every default.

1. Calculate the anticipated savings at each threshold using the number of true positives, the savings due to prevented default (estimated using last month’s bill), and the effectiveness rate of counseling:

total\_savings[counter] = np.sum(

true\_pos.astype(int)

\* X\_test\_all[:,5]

\* effectiveness

)

1. Increment the counter:

counter += 1

Steps 5 through 13 should be run as a for loop in one cell in the Jupyter notebook. Afterwards, the net savings for each threshold can be calculated as the savings minus the cost.

1. Calculate net savings for all the thresholds by subtracting the savings and cost arrays:

net\_savings = total\_savings - total\_cost

Now, we're in a position to visualize how much money we might help our client save by providing counseling to the appropriate account holders. Let's visualize this.

1. Plot the net savings against the thresholds as follows:

mpl.rcParams['figure.dpi'] = 400

plt.plot(thresholds, net\_savings)

plt.xlabel('Threshold')

plt.ylabel('Net savings (NT$)')

plt.xticks(np.linspace(0,1,11))

plt.grid(True)

The resulting plot should look like this:

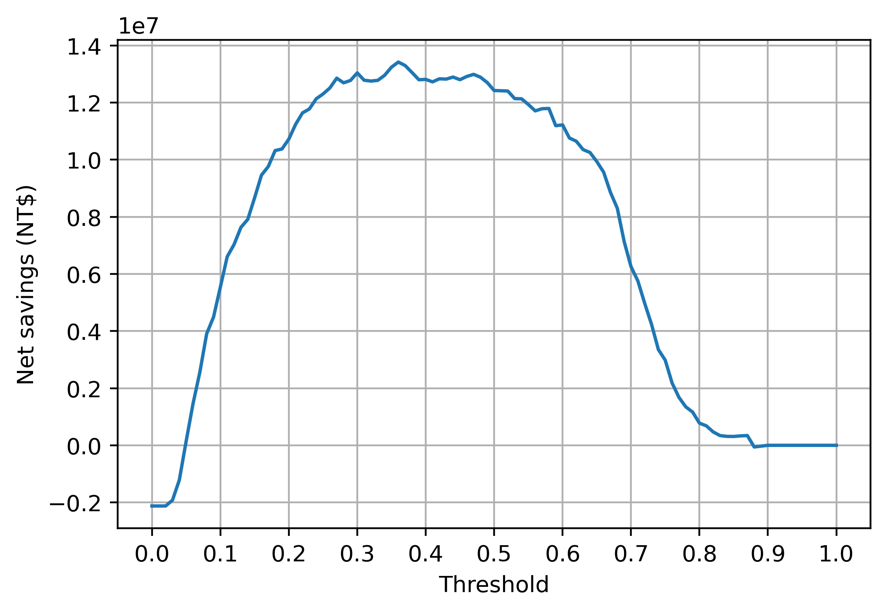


Figure 7.11: Plot of net savings versus thresholds

The plot indicates that the choice of threshold is important. While it will be possible to create net savings at many different values of the threshold, it looks like the highest net savings will be generated by setting the threshold somewhere in the range of about 0.25 to 0.5.

Let's confirm the optimal threshold for creating the greatest savings and see how much the savings are.

1. Find the index of the largest element of the net savings array using NumPy's argmax:

max\_savings\_ix = np.argmax(net\_savings)

1. Display the threshold that results in the greatest net savings:

thresholds[max\_savings\_ix]

The output should be as follows:

0.36

1. Display the greatest possible net savings:

net\_savings[max\_savings\_ix]

The output should be as follows:

13415710.0

We see that the greatest net savings occurs at a threshold of 0.36. The amount of net savings realized at this threshold is over NT$13 million, for this testing dataset of accounts. These savings would need to be scaled by the number of accounts served by the client, to estimate the total possible savings, assuming the data we are working with is representative of all these accounts.

Note, however, that the savings is about the same up to a threshold of about 0.5, as seen in Figure 7.11.

As the threshold increases, we are "raising the bar" for how risky a client must be, in order for us to contact them and offer counseling. Increasing the threshold from 0.36 to 0.5 means we would be only contacting riskier clients whose probability is > 0.5. This means contacting fewer clients, reducing the up-front cost of the program. Figure 7.11 indicates that we may be still able to create roughly the same amount of net savings, by contacting fewer people. While the net effect is the same, the initial expenditure on counseling will be smaller. This may be desirable to the client. We explore this concept further in the following activity.

Activity 6: Deriving Financial Insights

The raw materials of the financial analysis are completed. However, we can generate some additional insights from these results, to provide the client with more context around how the predictive model we built can generate value for them. In particular, we have looked at results for the testing set we reserved from model building. The client may have more accounts than those they supplied to us, as representative of their business. We should report to them results that could be easily scaled to however big their business is, in terms of number of accounts.

We can also help them understand how much this program will cost; while the net savings are an important number to consider, the client will have to fund the counseling program before any of these savings will be obtained. Finally, we will link the financial analysis back to standard machine learning model performance metrics.

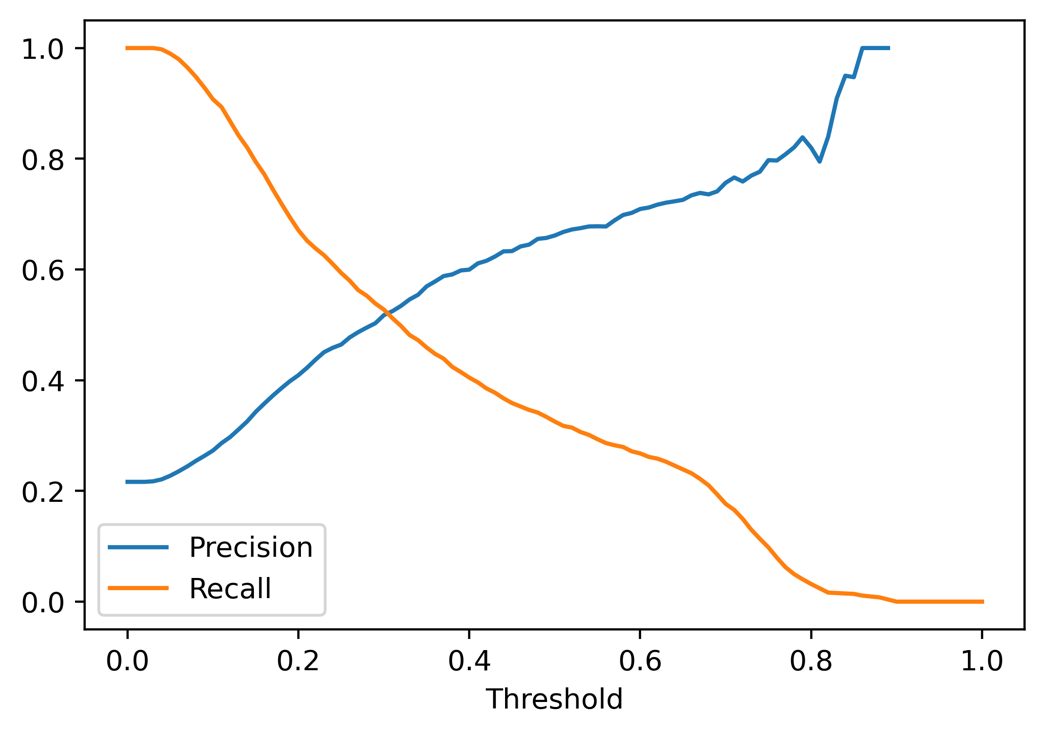
Once you complete the activity, you should be able to communicate the initial cost of the counseling program to the client, as well as obtain plots of precision and recall such as this:  


Figure 7.12: Expected precision-recall curve

This curve will be useful in interpreting the value created by the model at different thresholds.

Perform the following steps to complete the activity:

Note

The code and the resulting output for this activity have been loaded in a Jupyter Notebook that can be found here: <insert link>. Additional steps to prepare data for this activity, based on previous results in this chapter, have been added to the notebook.

1. Using the testing set, calculate the cost of all defaults if there were no counseling program.
2. Calculate by what percent the cost of defaults can be decreased by the counseling program.
3. Calculate the net savings per account at the optimal threshold, considering all accounts it might be possible to counsel.
4. Plot the net savings per account against the cost of counseling per account for each threshold.
5. Plot the fraction of accounts predicted as positive (this is called the "flag rate") at each threshold.
6. Plot a precision-recall curve for the testing data.
7. Plot precision and recall separately on the y-axis against threshold on the x-axis.

Note

The solution to this activity can be found on page <insert page number>.

Final Thoughts on Delivering the Predictive Model to the Client

We have now completed modeling activities and also created a financial analysis to indicate to the client how they can use the model. While we have created the essential intellectual contributions that are the data scientists' responsibility, it is necessary to agree with the client on the form in which all these contributions will be delivered.

A key contribution is the predictive capability embodied in the trained model. Assuming the client can work with the trained model object we created with XGBoost, this model could be saved to disk as we’ve done and sent to the client. Then, the client would be able to use it within their workflow. This pathway to model delivery may require the data scientist to work with engineers in the client’s organization, to deploy the model within the client’s infrastructure.

Alternatively, it may be necessary to express the model as a mathematical equation (i.e. logistic regression) or a set of if-then statements (i.e. decision tree or random forest) that the client could use to implement the predictive capability in SQL. While expressing random forests in SQL code is cumbersome due to the possibility of having many trees with many levels, there are software packages which will create this representation for you from a trained scikit-learn model (for example, <https://pypi.org/project/SKompiler/>).

Cloud Platforms for Model Development and Deployment

In this book we used scikit-learn and the XGBoost package to build predictive models, locally on our computers. Recently, cloud platforms such as AWS (Amazon Web Services) have made machine learning capabilities available through offerings such as Amazon SageMaker. SageMaker includes a version of XGBoost, which you can use to train models with similar syntax to what we’ve done here. Subtle differences may exist in the implementation of model training between the methods shown in this book and the Amazon distribution of SageMaker, and the reader is encouraged to check their work every step of the way to make sure their results are as intended. For example, fitting an XGBoost model using early stopping may require additional steps in SageMaker to ensure the trained model uses the best iteration for predictions, as opposed to the last iteration when training stopped.

Cloud platforms such as AWS are attractive because they may greatly simplify the process of integrating a trained machine learning model into a client’s technical stack, which in many cases is likely already built on a cloud platform.

Before using the model to make predictions, the client would need to ensure that the data was prepared in the same way it was for the model building we have done. For example, the removal of samples with values of 0 for all the features and the cleaning of the EDUCATION and MARRIAGE features would have to be done in the same way we demonstrated earlier in this chapter. Alternatively, there are other possible ways to deliver model predictions, such as an arrangement where the client delivers features to the data scientist and receives the predictions back.

Another important consideration for the discussion of deliverables is: what format should the predictions be delivered in? A typical delivery format for predictions from a binary classification model, like that we've created for the case study, is to rank accounts on their predicted probability of default. The predicted probability should be supplied along with the account ID and whatever other columns the client would like. This way, when the call center is working their way through the list of account holders to offer counseling to, they can contact those at highest risk for default first and proceed to lower priority account holders as time and resources allow. The client should be informed of which threshold to use for predicted probabilities, to result in the highest net savings. This threshold would represent the stopping point on the list of account holders to contact, if it is ranked on predicted probability of default.

Model Monitoring

Depending on how long the client has engaged the data scientist for, it is always beneficial to monitor the performance of the model over time, as it is being used. Does predictive capability remain the same or degrade over time? When assessing this, it's important to keep in mind that if account holders are receiving counseling, their probability of default would be expected to be lower than the predicted probability indicates, due to the intended effects of the new counseling program. For this reason, and to test the effectiveness of the counseling program, it is good practice to reserve a randomly chosen portion of account holders who will not receive any counseling, regardless of credit default risk. This group will be known as the control group and should be small compared to the rest of the population who receives counseling, but large enough to draw statistically significant inferences from.

While it's beyond the scope of this book to go in to details about how to design and use a control group, suffice to say here that model predictive capability could be assessed on the control group since they have received no counseling, similar to the population of accounts the model was trained on. Another benefit of a control group is that the rate of default, and financial loss due to defaults, can be compared to those accounts that received the model-guided counseling program. If the program is working as intended, the accounts receiving the counseling program will have a lower rate of default and a smaller financial loss due to default. The control group can provide evidence that the program is, in fact, working.

Advanced Modeling Technique for Selective Treatments: Uplift Modeling

When a business is considering selectively offering a costly treatment to its customers, such as the counseling program, a technique known as uplift modeling can be useful. Uplift modeling seeks to determine, on an individual basis, how effective treatments are. We made a blanket assumption that the phone counseling treatment is 70% effective across customers on average. However, it may be that the effectiveness varies by customer; some customers are more receptive and others less so. For more information on uplift modeling, see <https://www.steveklosterman.com/uplift-modeling/>.

A relatively simple way to monitor a model implementation is to see if the distribution of model predictions is changing over time, as compared to the population used for model training. We plotted the histogram of predicted probabilities on the test set in Figure 7.2. If the shape of the histogram of predicted probabilities changes substantially, it may be a sign that the features have changed, or that the relationship between the features and response has changed and the model may need to be re-trained or rebuilt. To quantify changes in distributions, the interested reader is encouraged to consult a statistics resource for the topics of the chi-squared goodness-of-fit test or the Kolmogorov-Smirnov test. Changing distributions of model predictions may also become evident if the proportion of accounts predicted to default, according to a chosen threshold, changes in a noticeable way.

Ethics in Predictive Modeling

The question of whether a model makes fair predictions has received increased attention as machine learning has expanded in scope to touch most modern businesses. Fairness may be assessed on the basis of whether a model is equally skillful at making predictions for different members of protected classes, for example different gender groups.

In this book we took the approach of removing gender from being considered as a feature for the model. However, it may be that other features can effectively serve as a proxy for gender, so that a model may wind up producing biased results for different gender groups, even though gender was not used as a feature. One simple way to screen for the possibility of such bias is to check if any of the features used in the model have a particularly high association with a protected class, for example by using a t-test. If so, it may be better to remove these features from the model.

How to determine whether a model is fair, and if not, what to do about it, is the subject of active research. The reader is encouraged to become familiar with efforts such as AI Fairness 360 (<https://aif360.mybluemix.net/>) that are making tools available to improve fairness in machine learning. Before embarking on work related to fairness, it’s important to understand from the client what the definition of fairness is, as this may vary by geographic region due to different laws in different countries, as well as the specific policies of the client’s organization.

Summary

In this chapter, you learned several analysis techniques to provide insight into model performance, such as decile and equal-interval charts of default rate by model prediction bin, as well as how to investigate the quality of model calibration. It’s good to derive these insights, as well as calculate metrics like the ROC AUC, using the model test set, since this is intended to represent how the model might perform in the real world on new data.

We also saw how to go about conducting a financial analysis of model performance. While we left this to the end of the book, an understanding of the costs and savings going along with the decisions to be guided by the model should be understood from the beginning of a typical project. These allow the data scientist to work toward a tangible goal in terms of increased profit or savings. A key step in this process, for binary classification models, is to choose a threshold of predicted probability at which to declare a positive sample, so that the profits or savings due to model-guided decision making are maximized.

Finally, we considered tasks related to delivering and monitoring the model, including the idea of establishing a control group to monitor model performance and test the effectiveness of any programs guided by model output. The structure of control groups and model monitoring strategies will be different from project to project, so you will need to determine the appropriate course of action in each new case. To further your knowledge of using models in the real world, you are encouraged to continue studying topics such as experimental design, cloud platforms such as AWS that can be used to train and deploy models, and issues with fairness in predictive modeling.

Congratulations, you have now completed the project and are ready to deliver your findings to the client! Along with trained models saved to disk, or other data products or services you may provide to the client, you will probably also want to create a presentation, typically a slide show, detailing your progress. Contents of such presentations usually include a problem statement, results of data exploration and cleaning, a comparison of the performance of different models you built, model explanations such as those using SHAP values, and the financial analysis which shows how valuable your work is. As you craft presentations of your work, it's usually better to tell your story with pictures as opposed to a lot of text. We've demonstrated many visualization techniques throughout the book that you can use to do this, and you should continue to explore ways to depict data and modeling results.

Always be sure to ask the client which specific things they may want to have in a presentation and be sure to answer all their questions. When a client sees that you can create value for them in an understandable way, you have succeeded.

Solution

Chapter 6: Imputation of Missing Data, Financial Analysis,   
and Delivery to Client

Activity 6: Deriving Financial Insights

1. Using the testing set, calculate the cost of all defaults if there were no counseling program.

Use this code for the calculation:

cost\_of\_defaults = np.sum(y\_test\_all \* X\_test\_all[:,5])

cost\_of\_defaults

The output should be:

60587763.0

1. Calculate by what percent the cost of defaults can be decreased by the counseling program.

The potential decrease in cost of default is the greatest possible net savings of the counseling program, divided by the cost of all defaults in the absence of a program:

net\_savings[max\_savings\_ix]/cost\_of\_defaults

The output should be:

0.2214260658542551

Results indicate that we can decrease the cost of defaults by 22% using a counseling program, guided by predictive modeling.

1. Calculate the net savings per account (considering all accounts it might be possible to counsel) at the optimal threshold.

Use this code for the calculation:

net\_savings[max\_savings\_ix]/len(y\_test\_all)

The output should be:

2259.2977433479286

Results like these help the client scale the potential amount of savings they could create with the counseling program, to as many accounts as they serve.

1. Plot the net savings per account against the cost of counseling per account for each threshold.

Create the plot with this code:

plt.plot(total\_cost/len(y\_test\_all),

net\_savings/len(y\_test\_all))

plt.xlabel('Upfront investment: cost of counselings per account (NT$)')

plt.ylabel('Net savings per account (NT$)')

The resulting plot should appear like this:

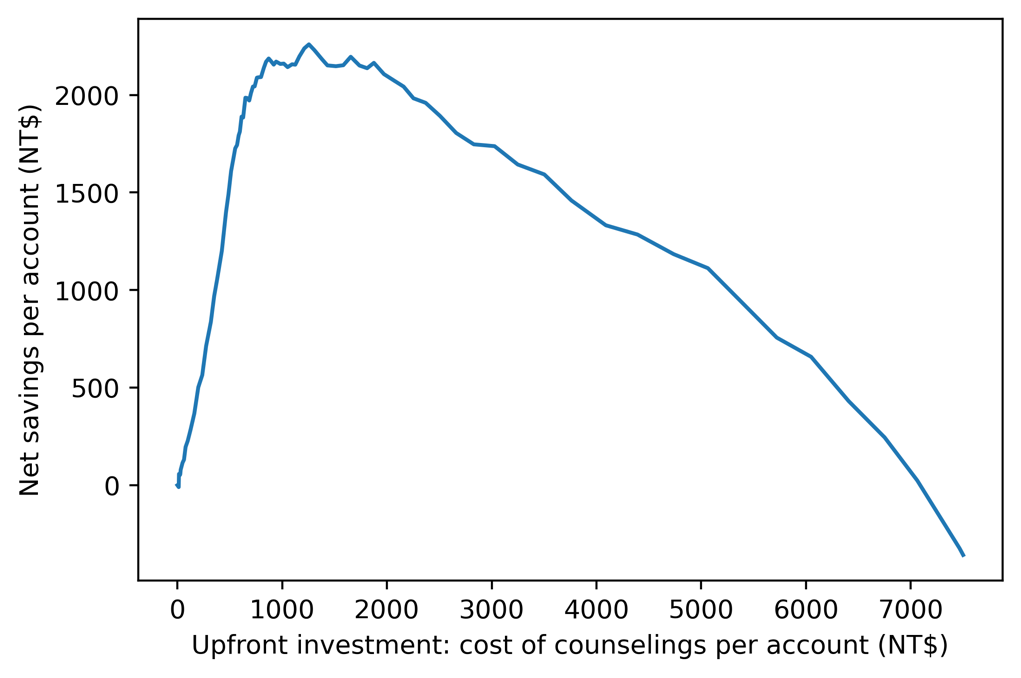


Figure 7.A1: The initial cost of the counseling program needed to achieve   
a given amount of savings

This indicates how much money the client needs to budget to the counseling program in a given month, to achieve a given amount of savings. It looks like the greatest benefit can be created by budgeting up to about NT$1300 per account (you could find the exact budgeted amount corresponding to maximum net savings using np.argmax). However, net savings are relatively flat for upfront investments between NT$1000 and 2000, being lower outside that range. The client may not actually be able to budget this much to the program. However, this graphic gives them evidence to argue for a larger budget if they need to.

This result corresponds to our graphic from the previous exercise. Although we've shown the optimal threshold is 0.36, it may be fine for the client to use a higher threshold up to about 0.5, thus making fewer positive predictions, offering counseling to fewer account holders, and having a smaller upfront program cost. Figure 7.A1 shows how this plays out in terms of cost and net savings per account.

1. Plot the fraction of accounts predicted as positive (this is called the "flag rate") at each threshold.

Use this code to plot the flag rate against the threshold:

plt.plot(thresholds, n\_pos\_pred/len(y\_test\_all))

plt.ylabel('Flag rate')

plt.xlabel('Threshold')

The plot should appear as follows:

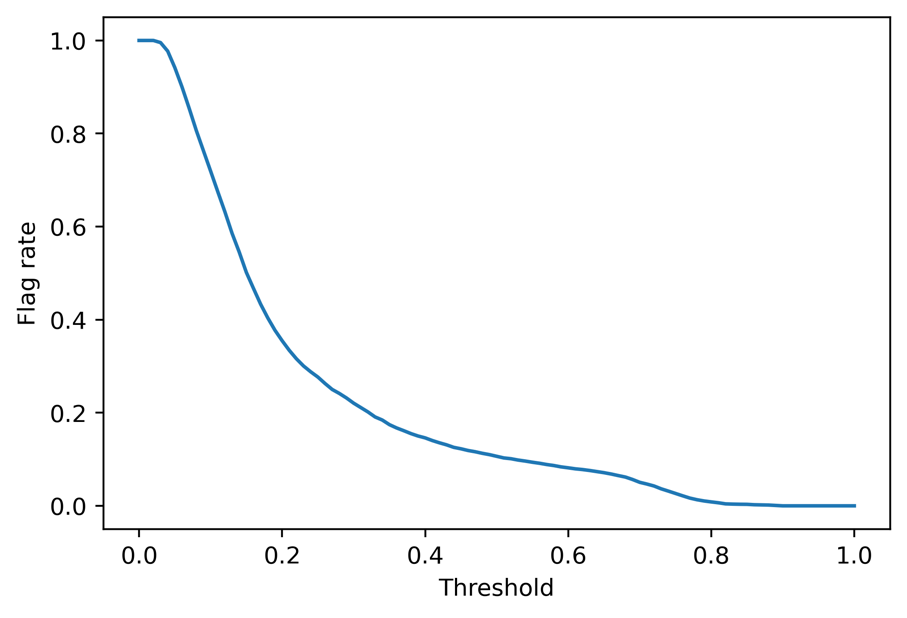


Figure 7.A2: Flag rate against threshold for the credit counseling program

This plot shows the fraction of people who will be predicted to default, and therefore recommended outreach at each threshold. It appears that at the optimal threshold of 0.36, only about 20% of accounts will be flagged for counseling. This shows how using a model to prioritize accounts for counseling can help focus on the right accounts and reduce wasted resources. Higher thresholds, which may result in nearly-optimal savings up to a threshold of about 0.5 as shown in Figure 7.11, result in lower flag rates.

1. Plot a precision-recall curve for the testing data using the following code:

plt.plot(n\_true\_pos/sum(y\_test\_all),

np.divide(n\_true\_pos, n\_pos\_pred))

plt.xlabel('Recall')

plt.ylabel('Precision')

The plot should look like this:

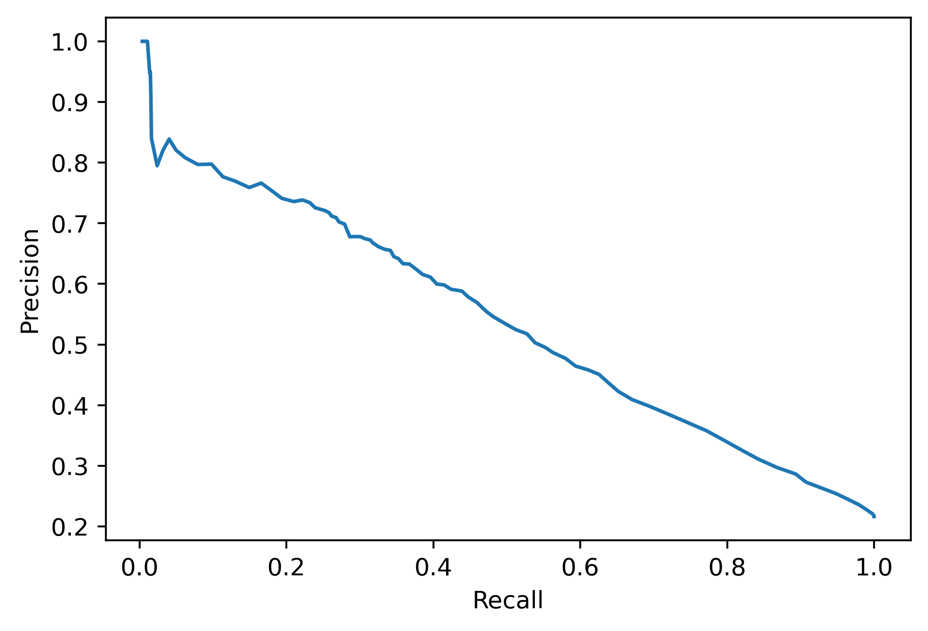


Figure 7.A3: Precision-recall curve

Figure 7.A3 shows that in order to start getting a true positive rate (that is, recall) much above 0, we need to accept a precision of about 0.8 or lower.

Precision and recall have a direct link to the cost and savings of the program: the more precise our predictions are, the less money we are wasting on counseling due to incorrect model predictions. And, the higher the recall, the more savings we can create by successfully identifying accounts that would default. Compare the code in this step to the code used to calculate cost and savings in the previous exercise to see this.

To see the connection of precision and recall with the threshold used to define positive and negative predictions, it can be instructive to plot them separately.

1. Plot precision and recall separately on the y-axis against threshold on the x-axis.

Use this code to produce the plot:

plt.plot(thresholds, np.divide(n\_true\_pos, n\_pos\_pred),

label='Precision')

plt.plot(thresholds, n\_true\_pos/sum(y\_test\_all),

label='Recall')

plt.xlabel('Threshold')

plt.legend()

The plot should appear as follows:

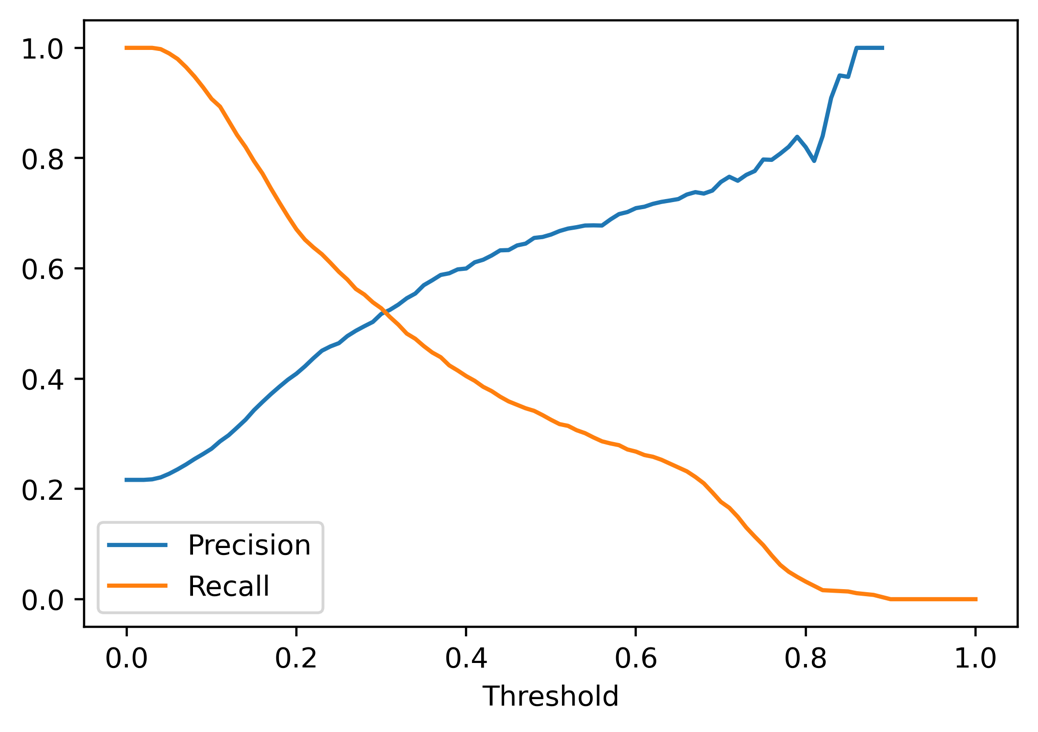


Figure 7.A4: Precision and recall plotted separately against threshold

This plot sheds some light on why the optimal threshold turned out to be 0.36. While the optimal threshold also depends on the financial analysis of costs and savings, we can see here that the steepest part of the initial increase in precision, which represents the correctness of positive predictions and is therefore a measure of how cost-effective the model-guided counseling can be, happens up to a threshold of about 0.36.