Module 12: Classification and K-Nearest Neighbors

Quick Reference Guide

Learning Outcomes:

1. Differentiate between regression, classification, clustering, and time series analysis problems
2. Train a nearest neighbor classifier
3. Create a visualization of KNN decision boundaries as a function of k and T (threshold)
4. Assess the tradeoffs between classifier metrics
5. Use cross-validation to pick hyperparameters that optimize the metric of your choosing
6. Compare the accuracy and interpretability of a properly trained linear regression and a KNN regression model

Introduction to Classification

Classification is distinct from regression. In a **regression problem** the goal is to predict a real-valued outcome, given a set of features. In a **classification problem**, the goal is to predict the class to which a sample belongs, given a set of features. For example, given the contents of an email, is it spam or not spam? Or, given the pixels of an image of an animal, which animal is it?

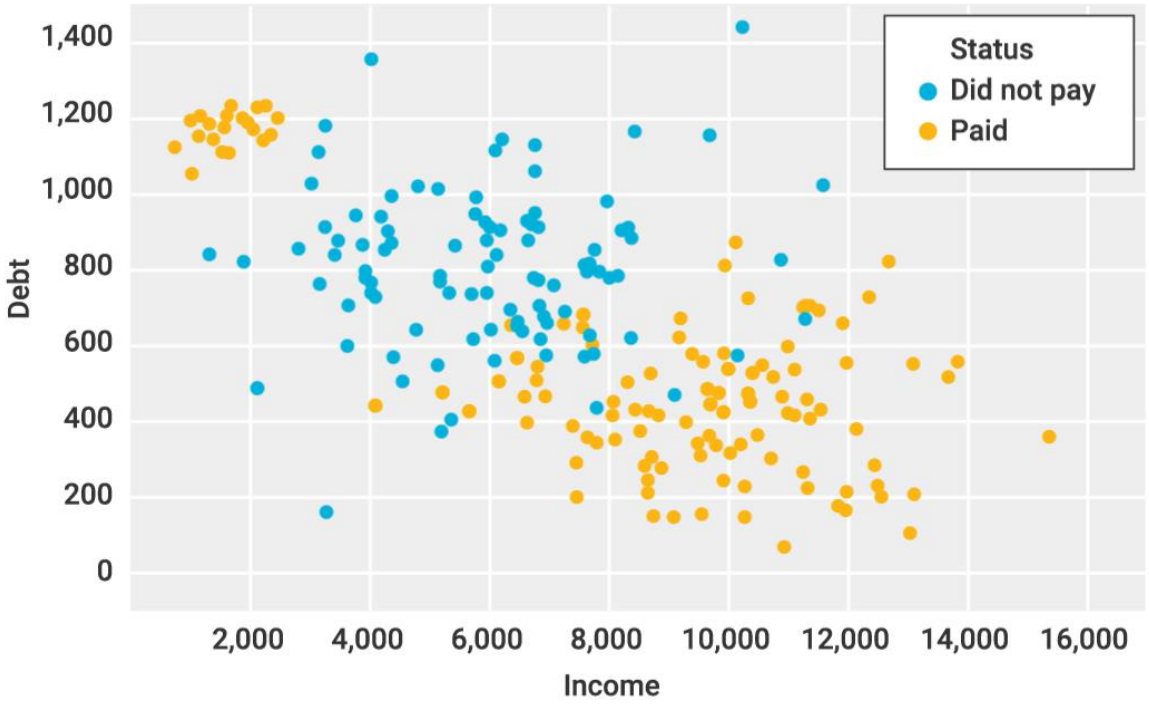
Creditors—like banks and credit card companies—often encounter regression problems: Given their credit history, how likely are potential customers to pay back a new loan?

The following table represents a fictitious past record of customers who have taken out a loan with a company. This can be considered as you try to build a model to predict the behavior of future customers.

	Index	Debt	Income	Status
0	191	365.25	12,281.95	Paid
1	176	682.45	7,849.78	Paid
2	99	877.54	4,278.69	Did not pay
3	131	122.27	8,984.55	Paid
4	154	220.76	8,893.86	Paid
5	283	603.26	3,845.26	Did not pay
6	25	185.73	9,685.38	Paid
7	83	610.00	8,315.45	Did not pay
8	115	685.29	5,923.50	Did not pay
9	62	845.69	2,364.19	Did not pay
10	49	185.66	9,700.58	Paid
11	18	1,178.32	6,900.09	Did not pay
12	143	1,264.61	1,700.63	Paid
13	39	1,068.37	985.28	Paid

Here, the status is whether or not in the last year, the customer made expected payments on their debts to that company. 'Paid' means all

payments were made and 'Did not pay' means that the customer defaulted on the debt. Note that the data only has two classes, 'Paid' and 'Did not pay.'



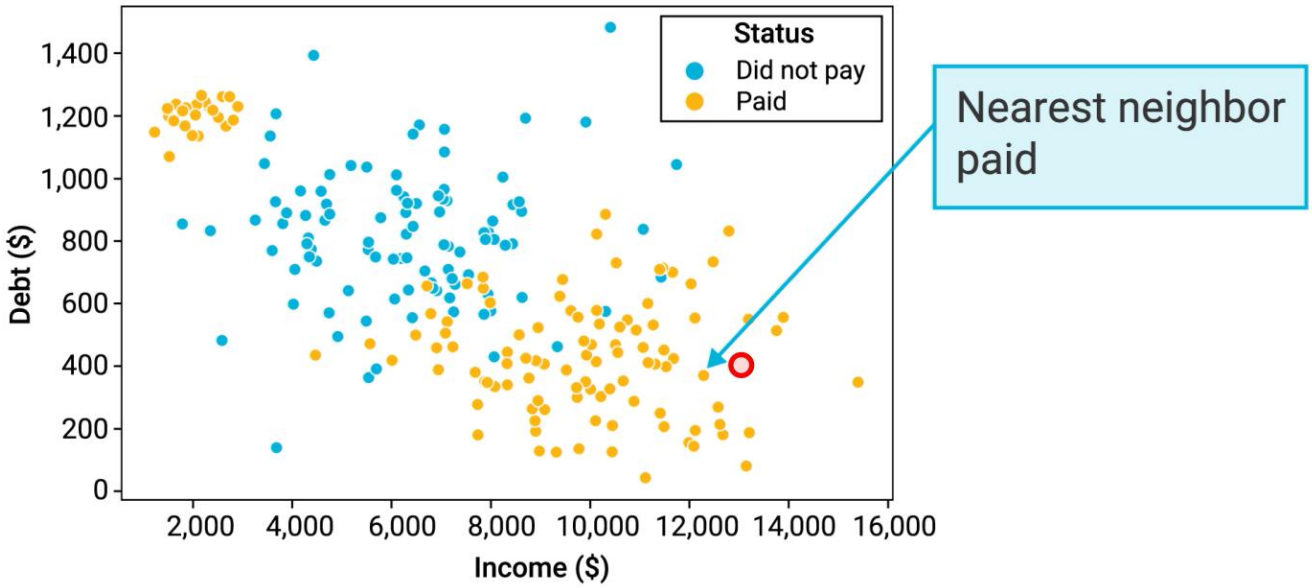
This plot graphically depicts the tabular data from before. The customer's income is on the x-axis and the customer's total debt is on the y-axis. The color indicates the status of the customer. Visually, three obvious groups emerge: two 'Paid' groups—one in the top-left corner and the other toward the bottom right—and a 'Did not pay' group roughly in the center of the plot.

Suppose a new customer arrives. This customer has an income of 13,000 and a total debt of 400. Without considering any particular algorithm, what do you predict the class of this new data point to be? Visually, this customer may seem firmly ensconced in one of the two 'Paid' regions. But how could you go about finding such patterns and classifying the customer's class by using an algorithm?

There are many approaches to classification. These include: Nearest neighbors, logistic regression, decision trees, Naive Bayes, random forests, support vector machines, perceptrons, and neural networks.

Nearest Neighbors in Scikit-Learn

The **nearest neighbors** classifier simply asks: Which data point in the training set is closest to the data point about which you are trying to make a prediction?



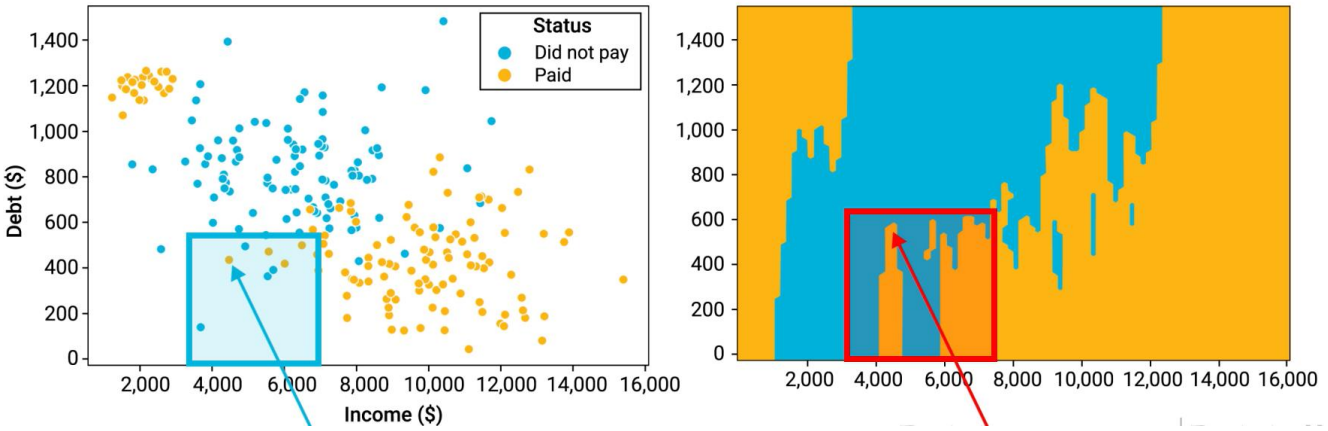
To use a nearest neighbors classifier in scikit-learn:

1. Instantiate an untrained model
2. Set a hyperparameter called n_neighbors = 1
3. Train the model, using the fit command

Like a regression estimator, a classification estimator also has a **predict function.** But it returns a label rather than a real value. The model does this by memorizing where all the existing data points are. When asked for a

prediction, it simply finds the closest, existing data point to that input and returns the appropriate prediction, such as Paid or Did not pay.

When visualizing the entire space of all hypothetical predictions, a **decision boundary** emerges as the edge between two classes—illustrated as colored regions.

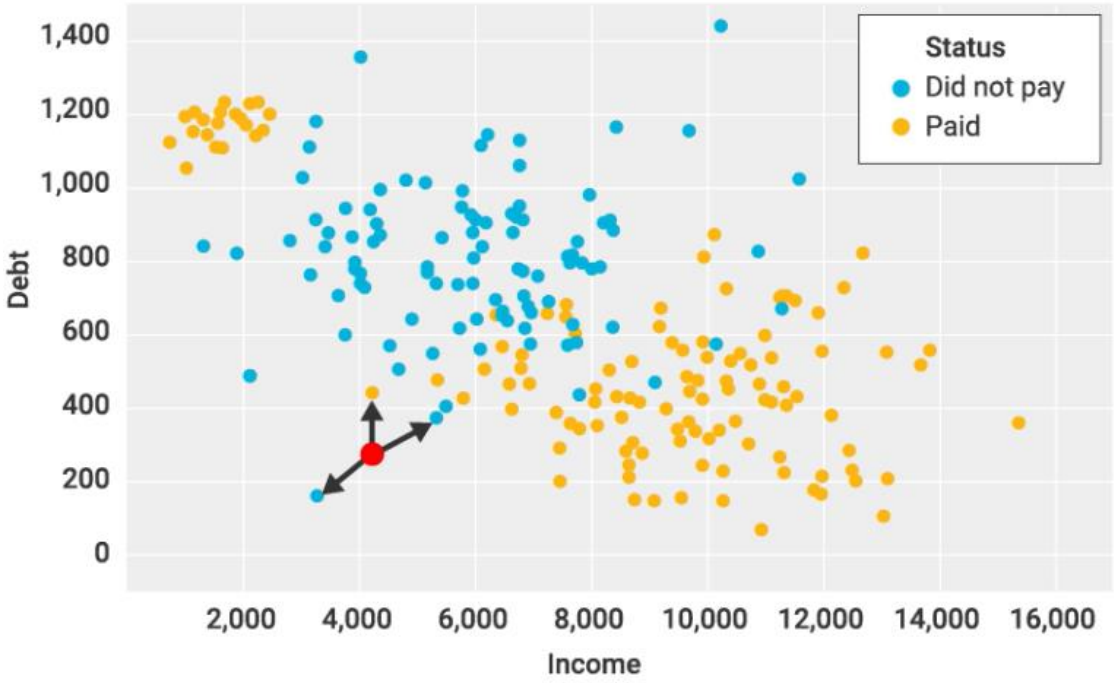


In this case the boundary is not a continuous surface, which suggests that the model may be too sensitive. Overly complicated boundaries—note, for example, the large shadow cast across the decision space by a single point—is indicative of **overfitting** where the model variance is high.

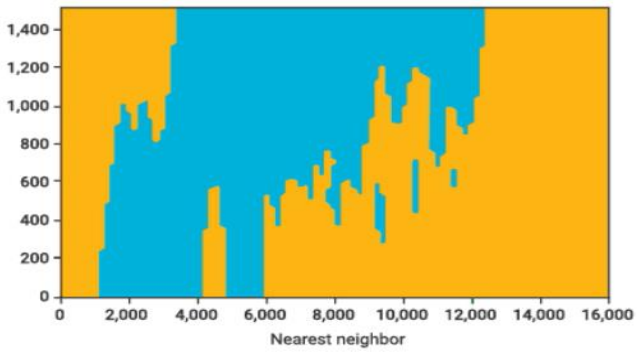
K-Nearest Neighbors

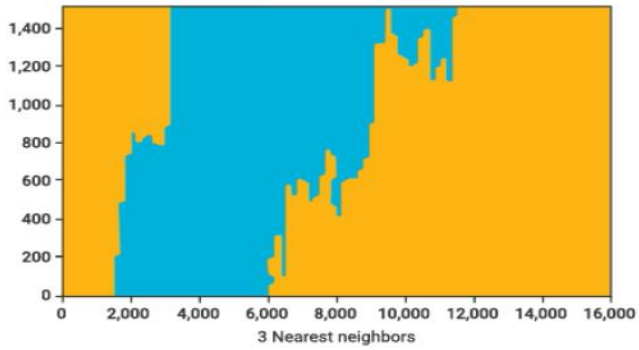
One approach to improving the nearest neighbor classifier is to consider multiple neighbors and doing a majority vote. To use, for example, three nearest neighbors in scikit-learn, simply set the n_neighbors hyperparameter to three when instantiating the classifier.

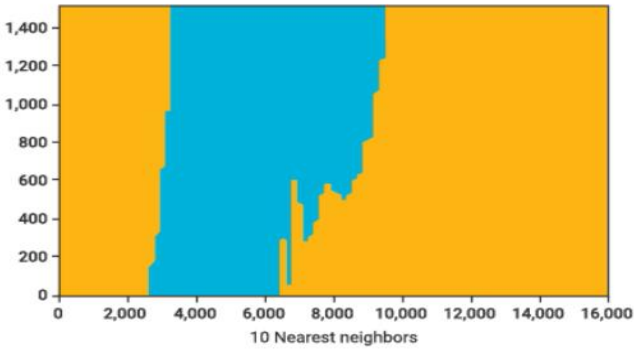
Now, you can increase the number of neighbors.

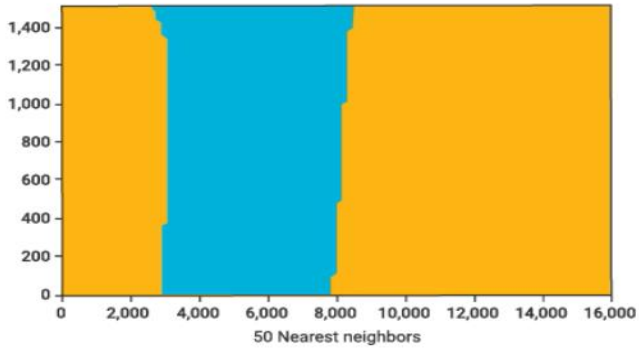


Consider the **decision boundaries** yielded by the 1, 3, 10, and 50 nearest neighbors models, respectively, for example.









As the number of neighbors increase, the region edges become smoother and smoother. The large shadow, cast by the outlier 'Paid' point in the bottom-left of the nearest neighbor plot shrinks—and eventually disappears—as the number of neighbors increases. Likewise, more and more of the decision space is classified as 'Paid.' Once n = 225, the entire decision space is classified as 'Paid,' since that is the most common class among the 225-point dataset.

There is a **tradeoff** in selecting k: For a very small k, the model is too complex and the variance is too high. But for very large k, the model is too simple. So, in other words, k is a complexity control hyperparameter, where complexity is inversely related to k. The maximum complexity is k = 1, and the minimum complexity is k = n.

Choosing K

When choosing k, k is a hyperparameter and thus the value you select for it can be consequential for your model.

Complexity increases with k, so when plotting the model's complexity against its error or variance, it makes sense to simply use –k for the x-axis. Defining the y-axis is trickier: Since the predictions and observations are not numerical quantities, using mean squared error (MSE) is no longer an option.

One option is to use the **misclassification rate**. In other words, the fraction of predictions that are incorrect.

As an example, consider the five-sample table shown.

Income	Debt	Status	ŷ
6,552.53	1,170.93	Did not pay	Did not pay
7,247.80	570.94	Did not pay	Did not pay
8,074.38	422.67	Did not pay	Paid
5,776.60	872.42	Did not pay	Did not pay
6,297.35	887.56	Did not pay	Did not pay

Here Status is what you are trying to predict and ŷ is the classifier's predictions. On this dataset, four out of the five predictions are correct. This translates to a misclassification rate of 20% or, equivalently, the **accuracy rate** is 80%. Note that the misclassification rate is always one minus accuracy.

You can compute the accuracy of a set of predictions by using the **sklearn.metrics.accuracy_score** function.

For example, calling this function on the data from the five-row table above, results in 0.8. The **get_misclassification_rate_for_k** function creates a nearest neighbors classifier, which considers the k-nearest neighbors. It then fits the classifier and returns the misclassification rate on the training set.

To set up an array of 50 different k's from one to 50, you can use the following code.

```
ks = range(1, 50)
errors = [get_misclassification_rate_for_k(k) for k in ks]
errors_and_ks = pd.DataFrame({'k' : ks, 'Misclassification Rate' : errors})
```

You can then use the **get_misclassification_rate_for_k** function to get the misclassification rate for each of the k's, as indicated in the table shown.

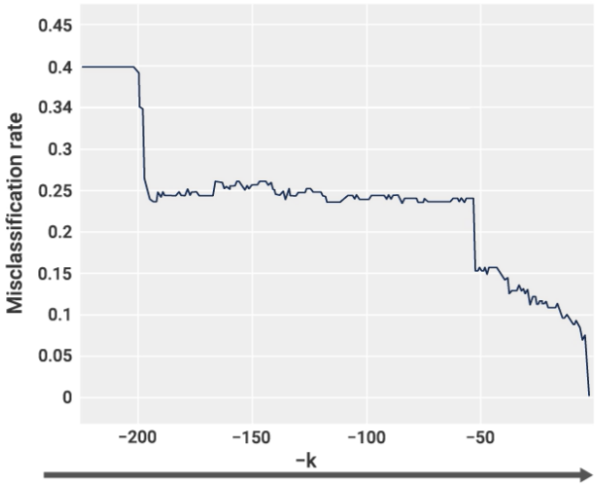
k	Misclassification rate
1	0.000000
2	0.062222
3	0.084444
4	0.075556
5	0.093333

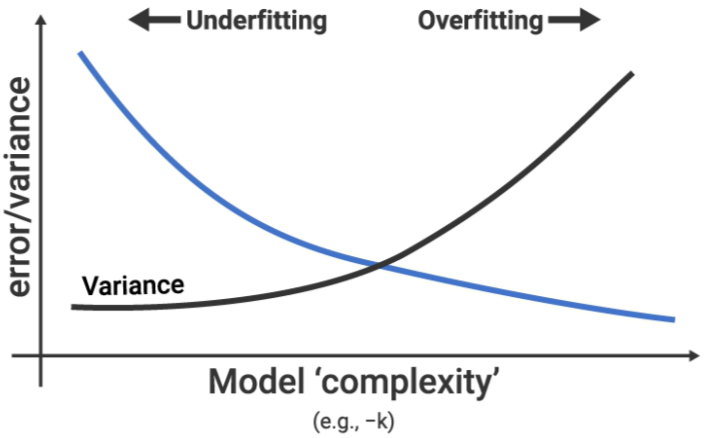
Note that for k = 1, the misclassification rate on the training set is zero. Since a classifier uses the training point's own class as its prediction, the prediction for any training point will always be correct.

As the value of k increases:

- The misclassification rate on the training set also goes up
- The boundaries get smoother
- The number of classification errors on the training set increases

To best match a visual plot of the misclassification rate versus k with the error versus complexity diagram introduced earlier on, plot –k on the x-axis.





In this dataset:

- 125 customers paid (55.5%)
- 100 customers did not pay (44.4%)
- For k = 225, the model will predict Paid for every point
- For k = 0, the misclassification rate is 0

Note that as the model complexity increases, the misclassification rate decreases. For low complexity models, the error is high, dropping as the model complexity increases. Note that the improvement is not monotonic.

Finding the best k

One method for choosing the best k is to apply **cross-validation**. In scikit-learn, you can use **GridSearchCV** to identify the k with the lowest validation error. Recall that **GridSearchCV** tries to maximize the score. Therefore, to minimize the error, or misclassification rate, you have to maximize the accuracy. So, for example, you can apply five-fold cross-validation with GridSearchCV to find the best n_neighbors for k's between one and 225 with 225 being the number of samples in the dataset.

After fitting the model, you ask the model_finder object for its best estimator. In this instance, it has trained a model with n_neighbors = 3, with 3 being the optimal value. So that means the cross-validation accuracy is maximized for k = 3.

For very large k's the test fails, returning not a number (NaN). This is because it does not make sense to have a k greater than the number of available neighbors.

Working with GridSearchCV results

To create a plot that is comparable to the familiar cartoon plot of validation error versus complexity, you need to:

- Plot –k on the x-axis, instead of k
- Plot the misclassification rate, one minus accuracy, on the y-axis

Predict_proba and Decision Thresholds

Classifiers provide a prediction about the most likely class. Scikit-learn models provide a function called **predict_proba**, which:

- Returns predictions about a class of a given sample
- Returns information about the level of confidence in the model

For example, you can call predict_proba() to make a prediction.

model.predict_proba(ten_random_rows[["Income", "Debt"]])

Consider the following set of ten random samples and the results of a 10-nearest neighbors classifier.

Income	Debt	Prediction
2,052.38	1,200.14	Paid
5,536.69	791.12	Did not pay
8,349.86	335.78	Paid
2,216.65	1,242.31	Did not pay
2,575.43	477.58	Paid
7,275.85	659.38	Did not pay
7,093.72	500.13	Did not pay
11,278.11	403.40	Paid
6,711.90	654.25	Did not pay
11,084.64	454.15	Paid

The result of calling predict_proba() is an array consisting of ten rows.

```
array([[0. , 1. ],
       [0.9, 0.1],
       [0.3, 0.7],
       [0. , 1. ],
       [0.3, 0.7],
       [0.7, 0.3],
       [0.5, 0.5],
       [0. , 1. ],
       [0.6, 0.4],
       [0.1, 0.9]])
```

Here, the first column of the array is an estimate of the probability that the sample belongs to class 0. The second column is the model's estimate of the probability that the sample belongs to class 1. For scikit-learn models, the 0th class is the class that comes earlier in alphabetical order. And so, since 'Did not pay' comes before 'Paid' in alphabetical order, 'Did not pay' is the 0th class.

Based on the first result in the array, [0. , 1.], the model is 100% certain that this person will pay back the loan. For the third sample, [0.3, 0.7], the model is only 70% certain that the person will pay back the loan.

For k-NearestNeighbors, the probabilities come from the votes from each of the nearest neighbors. If, for example, two of the ten nearest neighbors defaulted on their loans and eight of them paid, the model would be 80% certain that the person will pay back their loan. The behavior of each available training sample is used as a vote. Since there are ten nearest neighbors in this dataset, the probabilities are always multiples of 0.1.

Decision threshold

For a binary classifier, scikit-learn returns class 0 if $p \leq 0.5$. And class 1 only if $p > 0.5$. Thus, for a sample to belong to class 1, it must **exceed** the threshold of 0.5. Any ties go to class 0, which is 'Did not pay.'

In the example, the seventh sample returns 'Did not pay,' even though the model is 50% certain that the person will pay back their loan. That is because scikit-learn has established a **probability threshold** of 0.5. Note that the cutoff point of 0.5 is an arbitrary choice by scikit-learn. If you decide, for example, to set a probability threshold of 80%, only samples exceeding 0.8 will be considered as part of the 'Paid' class.

Classifier Metrics

Before, accuracy was used to assess the quality of a classifier. But accuracy is not always the right metric. Consider, for example, the year 2020 when COVID-19 first started spreading. Initially, when very few people were infected, a fake test that simply said 'not infected' would have had extremely high accuracy. If only 100,000 out of the 1.4 billion people in the world were infected, the accuracy of this bad test would be 1.399 billion out of 1.4 billion, or around 99.993%. This is an example of a situation with **imbalanced classes**. In other words, there are different numbers of samples from each class. In cases with imbalanced classes, accuracy does not necessarily give a good assessment of the usefulness of a model.

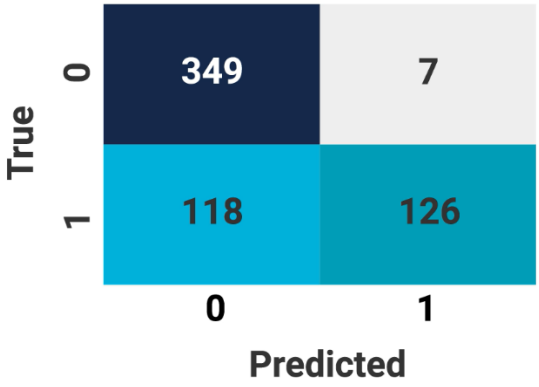
Confusion matrix

A confusion matrix paints a more thorough picture of classifier usefulness, retaining usefulness even for wildly imbalanced classes.

For a **binary classifier**, with only two classes, the confusion matrix provides a count of:

- True negative (TN)
- False positive (FP)
- False negative (FN)
- True positive (TP)

Returning to the COVID-19 example, consider its confusion matrix.



Here, the y-axis relates to whether or not someone is actually infected. And the x-axis whether or not the test predicts that they are infected.

True negatives are situations where the model correctly predicts that the patient is healthy. In this example, there are 349 of those. **False positives** are the seven instances where this model predicted the patient was sick, but they were not. **False negatives** are where the model predicted that the patient is healthy, but they were actually sick, there are 118 such cases in this example. **True positives** are where the model correctly predicted that a patient is sick, in this instance there are 126 such cases.

Summarizing a confusion matrix

Confusion matrices may be tricky to read and interpret. Some of the many ways in which to summarize a confusion matrix into a single entity include:

- Accuracy
- Precision
- Recall
- Specificity

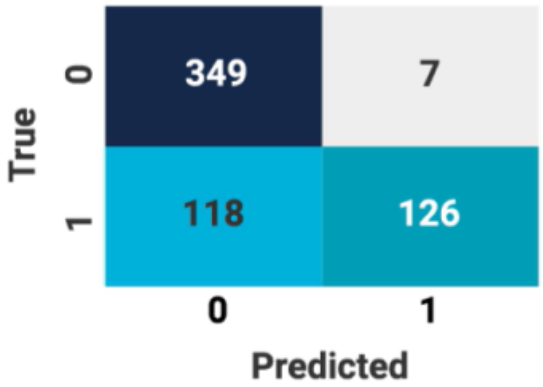
Accuracy measures how well the model's prediction correlates to reality. Calculate total accuracy by dividing the sum of TP and TN by the sum of TN, FP, FN and TP. For the COVID-19 example, the accuracy amounts to 79.1%.

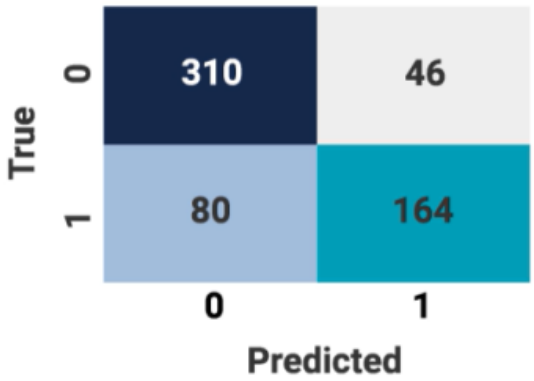
Precision measures how many of the people that tested positive actually have COVID-19. To calculate precision, simply divide TP by the sum of TP and FP. For this matrix the precision is 94.7%.

Recall measures how many of those with COVID-19 the model was able to correctly predict. Calculating recall involves dividing TP by the sum of TP and FN, which amounts to 51.6% in this case. Note that recall is often referred to as sensitivity.

Specificity measures how many of those that do not have COVID-19 the model was able to correctly predict. Calculate the specificity by dividing TN by the sum of TN and FP. In this example it is quite high, 98%.

Similar accuracies in models do not necessarily imply the similarity of other metrics. Consider, for example the following two models predicting whether a patient is sick.





While both models have an accuracy of around 79%, their other metrics are quite different. For example, model 1 has a precision of 126/126 + 7, or 95%, while model 2 has a precision of 164/164 + 46, or 78%. In other words, for predictions that a patient is sick, the first model, with 95% precision, is much more likely to be correct in its prediction.

Choosing a Classifier Metric

Since emphasis on a single metric can result in models that are not particularly useful, it is important to focus on the tradeoffs between metrics instead. One of the most commonly explored tradeoffs among these metrics, is between precision and recall. Consider the meaning of these terms in the context of a loan model:

- **Precision**: If you predict that a person will pay their loan, what is the chance that you are correct?
- **Recall**: If a person that will pay exists, what is the chance that you correctly identify them as someone who will pay?

In a business context, for example, both these metrics are important. High precision means that customers who receive loans are likely to pay them back. High recall means that potential quality customers are not overlooked.

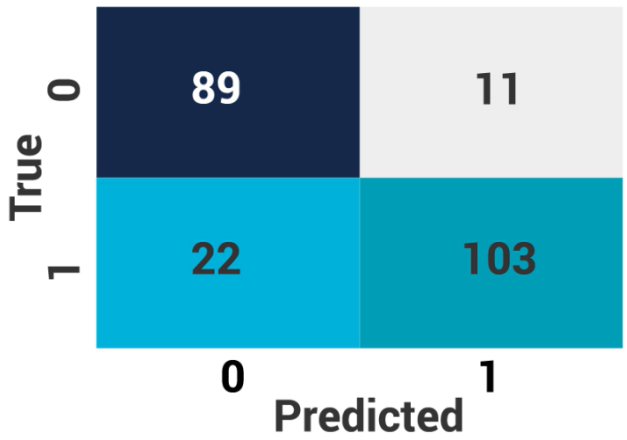
Optimistic classifiers have high recall but low precision. Revisiting the loan example: optimistic classifiers tend to think people will pay their loans. On the extreme end, a very optimistic classifier would only disqualify the most dangerous-looking customers. That would result in high recall because the model would correctly identify every customer who is likely to pay back their loan. But it would have low precision because it would treat many

dangerous customers as safe. This yields a high-risk situation where many loans are granted, but many of the customers default.

Pessimistic classifiers have a low recall but high precision. In the context of a loan, pessimistic classifiers tend to think people will not pay their loans. Extremely pessimistic classifiers, which only classify the safest customers as part of the 'Paid' class, will result in low recall missing a vast fraction of good customers. The precision would be high because pessimistic classifiers are unlikely to make a mistake when they approve someone as a good customer. That yields a low-risk, low-reward situation, where loans are only given to a small number of potentially profitable customers.

Adjusting the decision threshold

Recall that a probability threshold (T) defines the cutoff point which separates classes from one another. Consider the confusion matrix for a k-nearest neighbors model below, where k = 30 and T = 0.5.

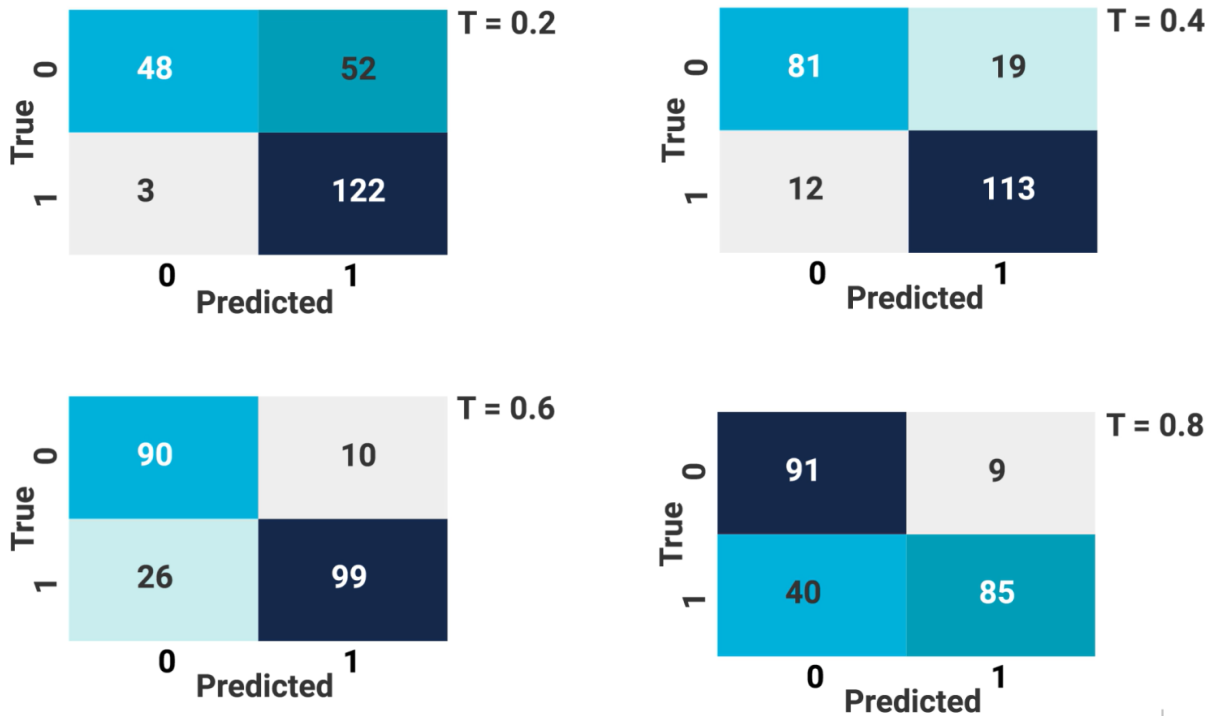


In the context of loans, as T increases, the model gets more pessimistic about customers' ability to pay and precision gets better. The model only

classifies data as 1 if it is very sure. Simultaneously, recall gets worse. The model will classify fewer and fewer data as class 1, in other words as 'Paid.'

The effect of increasing T on precision and recall

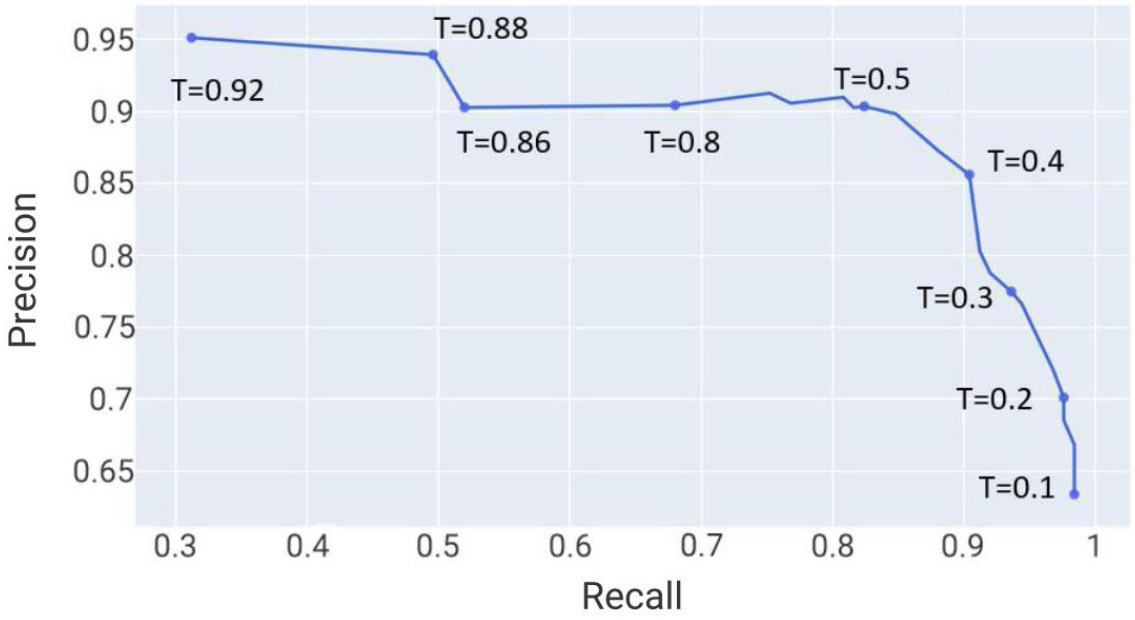
Consider the confusion matrices for the loan payback model with n_neighbors = 30, as T increases from 0.2 to 0.4 to 0.6 to 0.8.



As T increases from 0.2 to 0.4 to 0.6 to 0.8, the precision grows from 70% to 86%, to 91%, and slips down a bit to 90%. By contrast, the recall drops from 98% to 90% to 79% to 68%. For recall, the trend monotonically decreases. For precision, the dependence on T is not monotonic.

Scikit-learn provides a useful function called **precision_recall_curve** that is used to generate precision and recall values for various classifier thresholds.

You can plot the resulting values on a curve, using Plotly, to yield a curve.



Small T-values in the bottom left represent high recall and low precision. High T-values in the top left have low recall and high precision. Given such a curve for a classifier, you can select the optimal threshold T. Since tradeoffs may vary widely, the best choice of threshold is rooted in domain expertise.

Using a precision–recall curve

A visual representation of the precision–recall tradeoff can be useful in a number of ways. These include:

- Identifying a suitable T-value
- Comparing two classifiers independent of decision thresholds
- Summarizing the entire precision–recall curve for model comparison

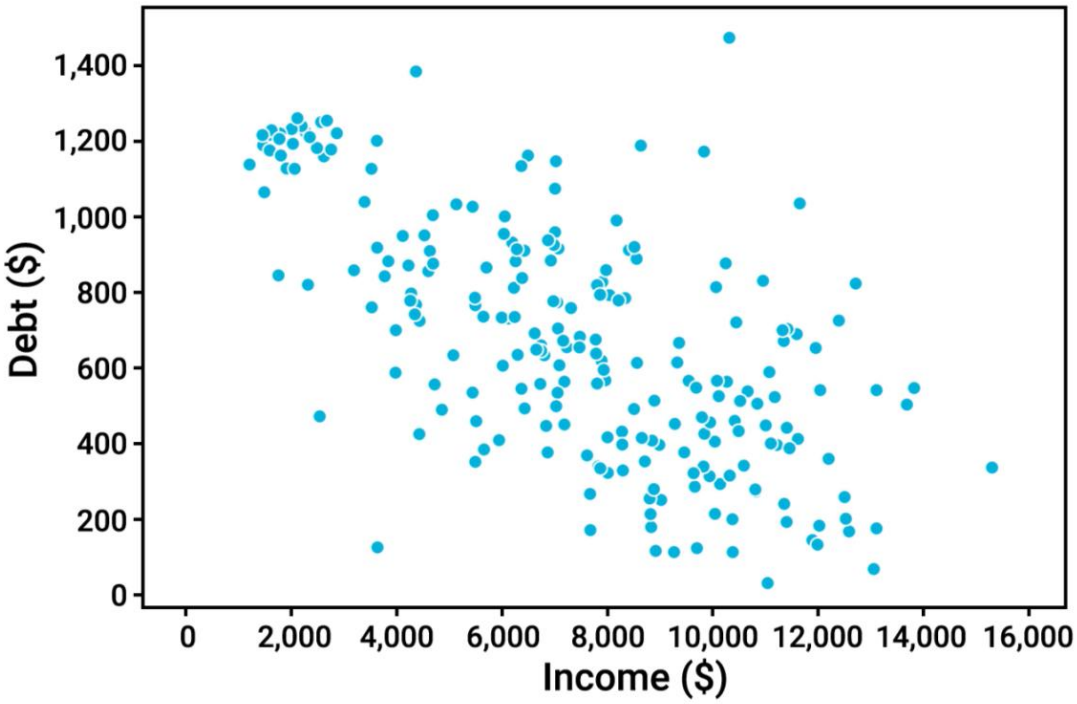
A common tradeoff in machine learning is between the recall and one minus specificity. This alternate tradeoff is also known as a **receiver operator characteristic curve (ROC curve)**. In the context of a ROC curve, the recall is often called the **true positive rate**, while one minus specificity is

called the **true negative rate**. Just like a precision–recall curve, you can take a ROC curve and summarize it by the area that it encloses. And, for a ROC curve, being closer to the top left is better.

GridSearchCV can compare two models based on the area under their ROC curve.

Regression Example: Using Nearest Neighbors for Regression

While it is not common in practice, you can use k-nearest neighbors to solve regression problems. Consider this plot, which is used to predict debt from income on the same dataset, as before .



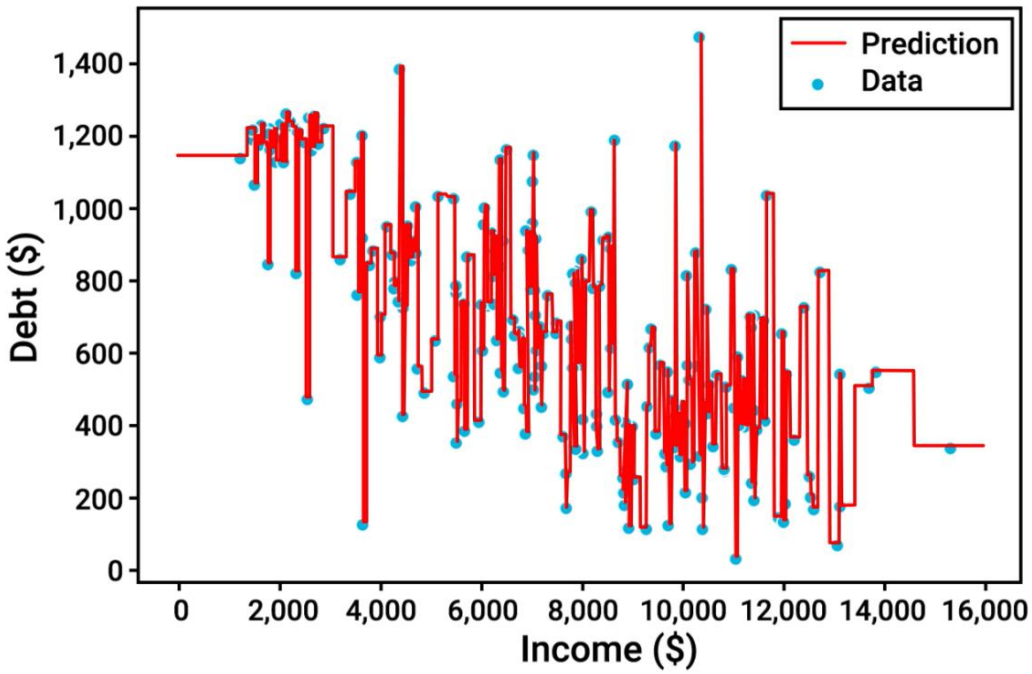
Suppose you have a new customer whose income is $15,800. What debt level will your model predict? Recall that a nearest-neighbors model identifies the single nearest neighbor where k = 1. In this plot, the nearest

neighbor is the rightmost point in the original dataset, which results in a predicted value of $343.

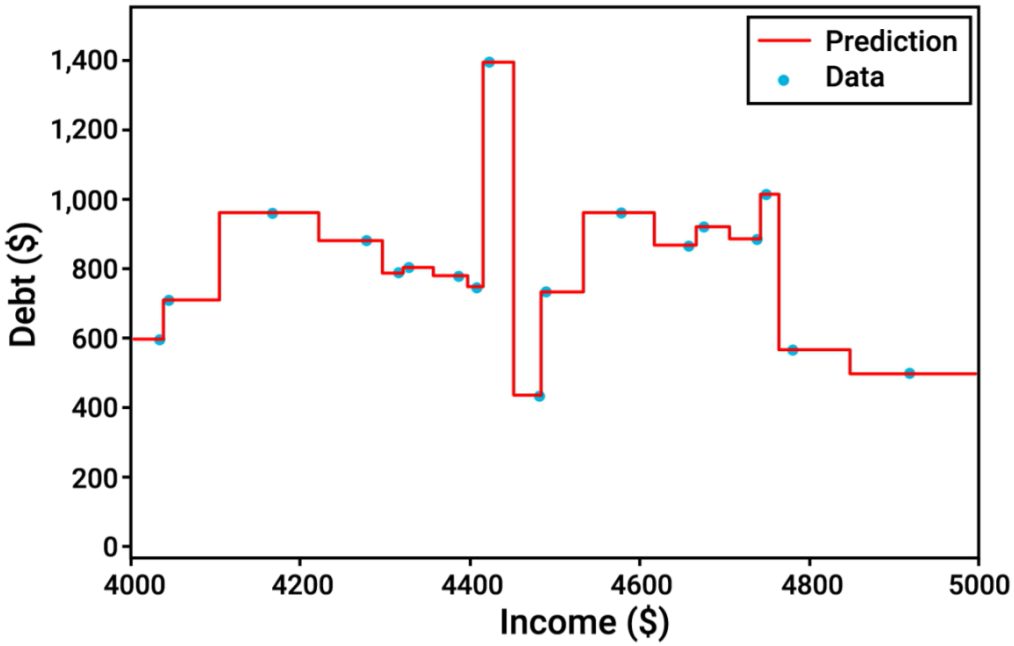
In scikit-learn you can use the **sklearn.neighbors.KNeighborsRegressor**. When using the KNeighborsRegressor, it is important to note the:

- Name of the class
- Names of the input and the output columns

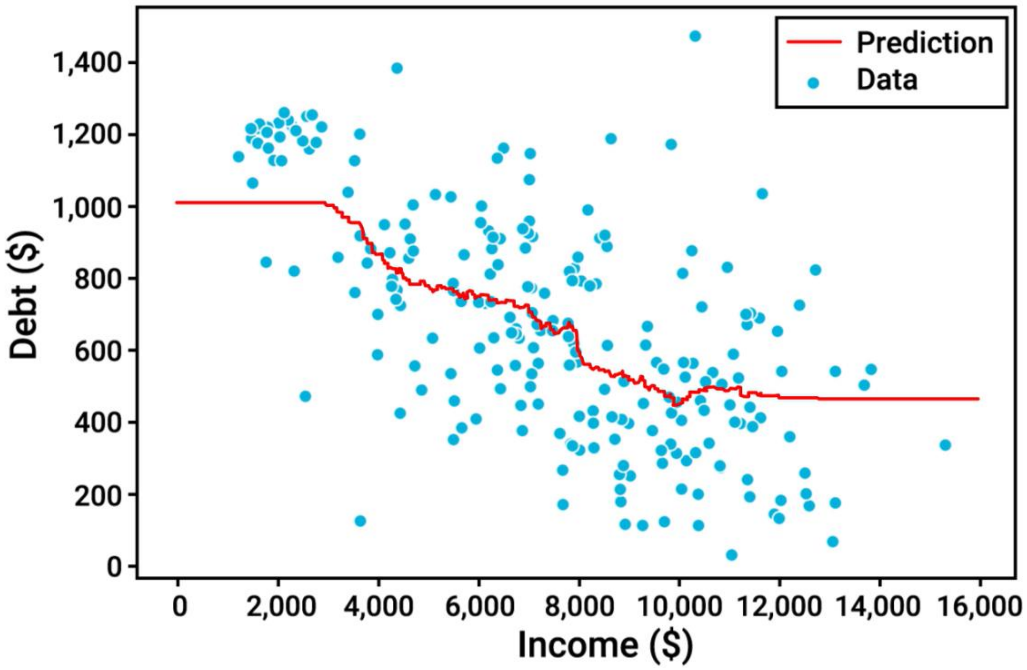
The regressor outputs a numerical value rather than a label. As is the case with linear regression curves, you can also plot the output of this model for k = 1.



In the case of particularly dense datasets, it may be difficult to identify a clear curve. Observe, for instance, the curve that emerges when you zoom in to the range between x = 4,000 and x = 5,000 on the plot.



Here the predictions start to resemble expectations. The output is always equal to the debt value for the closest input.



Just like before, increasing k smooths out the plot. Observe, for example, the output of the same model when k = 50.

As is the case with classification, behavior will become poor for very large values of k.

To select an **optimum k** for a **regression problem**:

- Use cross-validation
- Choose mean squared error (MSE) as your loss

Depending on the type of problem, identify a value of k that maximizes the metric of your choosing.

Refugee Resettlement and Prediction

To achieve impactful results with data science tools, you should always consider the specific business or policy environment. Understanding the underlying **theory of change** is key.

Consider the example from an academic paper that studies whether refugee resettlement could be done more effectively by using an algorithm to assign refugees to locations. The **goal** is to try and determine how to resettle refugees so they can have **the best chance of success** in their new country. The study focused on refugees arriving in the United States and Switzerland, two countries where there is good data available. This setting seems to lend itself to the use of a prediction model. The study requires the prediction of outcomes, such as the likelihood of a refugee being employed within a particular timeframe.

Here, the underlying theory of change dictates that you ask whether a refugee could be reassigned to another location **and** be more successful—not merely how successful they will be on average.

As it turns out, refugee resettlement is basically done randomly, subject to availability and population, but—importantly—not based on expected outcome. When the prediction problem is similarly turned into a question of matching, data science tools can be used to build a supervised model that predicts employment outcomes for each location based on the observable characteristics of individual refugees. The predictions can then be mapped into how refugee resettlement rules could be rewritten to adhere to the algorithmic predictions and in so doing maximize employment outcomes.

The results of the study are striking. Were refugee agencies to rely on an assignment algorithm, the prediction is that the U.S. could increase employment of refugees by 41%. In the Swiss setting, the assessment for long-run outcomes finds that employment three years later could be increased by 73% using algorithmic assignment. The efficacy and cost-effectiveness of this kind of approach is a dramatic improvement over far more costly interventions, such as retraining programs.

The use of standard prediction tools to estimate matches have many potential real-world applications. Consider, for example, a sales organization trying to assign different salespeople to certain types of leads. Ideally, you would want to go beyond simply predicting lead quality on average so you can allocate each salesperson to the best lead for them. Here it would be important to investigate randomization or near randomization.