This time, all but two of the original features were selected. Because we specified to select 40 features, some of the noise features are also selected. Let's take a look at the performance:

In[45]:

```
X_test_l1 = select.transform(X_test)
score = LogisticRegression().fit(X_train_l1, y_train).score(X_test_l1, y_test)
print("Test score: {:.3f}".format(score))
```

Out[45]:

```
Test score: 0.951
```

With the better feature selection, we also gained some improvements here.

Iterative Feature Selection

In univariate testing we used no model, while in model-based selection we used a single model to select features. In iterative feature selection, a series of models are built, with varying numbers of features. There are two basic methods: starting with no features and adding features one by one until some stopping criterion is reached, or starting with all features and removing features one by one until some stopping criterion is reached. Because a series of models are built, these methods are much more computationally expensive than the methods we discussed previously. One particular method of this kind is *recursive feature elimination* (RFE), which starts with all features, builds a model, and discards the least important feature according to the model. Then a new model is built using all but the discarded feature, and so on until only a prespecified number of features are left. For this to work, the model used for selection needs to provide some way to determine feature importance, as was the case for the model-based selection. Here, we use the same random forest model that we used earlier, and get the results shown in Figure 4-11:

In[46]:

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
             n_features_to_select=40)

select.fit(X_train, y_train)
# visualize the selected features:
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

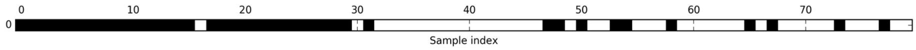

Figure 4-11. Features selected by recursive feature elimination with the random forest classifier model

The feature selection got better compared to the univariate and model-based selection, but one feature was still missed. Running this code also takes significantly longer than that for the model-based selection, because a random forest model is trained 40 times, once for each feature that is dropped. Let's test the accuracy of the logistic regression model when using RFE for feature selection:

In[47]:

```
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)

score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("Test score: {:.3f}".format(score))
```

Out[47]:

```
Test score: 0.951
```

We can also use the model used inside the RFE to make predictions. This uses only the feature set that was selected:

In[48]:

```
print("Test score: {:.3f}".format(select.score(X_test, y_test)))
```

Out[48]:

```
Test score: 0.951
```

Here, the performance of the random forest used inside the RFE is the same as that achieved by training a logistic regression model on top of the selected features. In other words, once we've selected the right features, the linear model performs as well as the random forest.

If you are unsure when selecting what to use as input to your machine learning algorithms, automatic feature selection can be quite helpful. It is also great for reducing the amount of features needed—for example, to speed up prediction or to allow for more interpretable models. In most real-world cases, applying feature selection is unlikely to provide large gains in performance. However, it is still a valuable tool in the toolbox of the feature engineer.

Utilizing Expert Knowledge

Feature engineering is often an important place to use *expert knowledge* for a particular application. While the purpose of machine learning in many cases is to avoid having to create a set of expert-designed rules, that doesn't mean that prior knowledge of the application or domain should be discarded. Often, domain experts can help in identifying useful features that are much more informative than the initial representation of the data. Imagine you work for a travel agency and want to predict flight prices. Let's say you have a record of prices together with dates, airlines, start locations, and destinations. A machine learning model might be able to build a decent model from that. Some important factors in flight prices, however, cannot be learned. For example, flights are usually more expensive during peak vacation months and around holidays. While the dates of some holidays (like Christmas) are fixed, and their effect can therefore be learned from the date, others might depend on the phases of the moon (like Hanukkah and Easter) or be set by authorities (like school holidays). These events cannot be learned from the data if each flight is only recorded using the (Gregorian) date. However, it is easy to add a feature that encodes whether a flight was on, preceding, or following a public or school holiday. In this way, prior knowledge about the nature of the task can be encoded in the features to aid a machine learning algorithm. Adding a feature does not force a machine learning algorithm to use it, and even if the holiday information turns out to be noninformative for flight prices, augmenting the data with this information doesn't hurt.

We'll now look at one particular case of using expert knowledge—though in this case it might be more rightfully called "common sense." The task is predicting bicycle rentals in front of Andreas's house.

In New York, Citi Bike operates a network of bicycle rental stations with a subscription system. The stations are all over the city and provide a convenient way to get around. Bike rental data is made public in an anonymized form and has been analyzed in various ways. The task we want to solve is to predict for a given time and day how many people will rent a bike in front of Andreas's house—so he knows if any bikes will be left for him.

We first load the data for August 2015 for this particular station as a `pandas DataFrame`. We resample the data into three-hour intervals to obtain the main trends for each day:

In[49]:

```
citibike = mglearn.datasets.load_citibike()
```