Automatic Feature Selection

Zhongjian Lin
University of Georgia

October 23, 2024

Features

- With so many ways to create new features, you might get tempted to increase the dimensionality of the data way beyond the number of original features.
- However, adding more features makes all models more complex, and so increases the chance of overfitting.
- When adding new features, or with high-dimensional datasets in general, it can be a good idea to reduce the number of features to only the most useful ones, and discard the rest. This can lead to simpler models that generalize better.

Strategies

There are three basic strategies: Univariate statistics, model-based selection and iterative selection. All three of these methods are supervised methods, meaning they need the target for fitting the model.

- In univariate statistics, we compute whether there is a statistically significant relationship between each feature and the target. Then the features that are related with the highest confidence are selected. In the case of classification, this is also known as analysis of variance (ANOVA).
- Model based feature selection uses a supervised machine learning model to judge the importance of each feature, and keeps only the most important ones. The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling.
- In iterative feature selection, a series of models is built, with varying numbers of features.

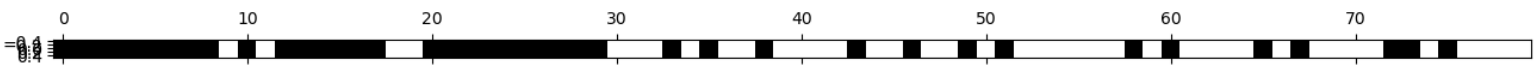
Univariate Statistics

- A key property of these tests are that they are univariate meaning that they only consider each feature individually. Consequently a feature will be discarded if it is only informative when combined with another feature.
- Univariate tests are often very fast to compute, and don't require building a model. On the other hand, they are completely independend of the model that you might want to apply after the feature selection.
- To use univariate feature selection in scikit-learn, you need to choose a test, usually either *f_classif* (the default) for classification or *f_regression* for regression, and a method to discard features based on the p-values determined in the test. All methods for discarding parameters use a threshold to discard all features with too high a p-values (which means they are unlikely to be related to the target). The methods differ in how they compute this threshold, with the simplest ones being *SelectKBest* which selects a fixed number k of features, and *SelectPercentile*, which selects a fixed percentage of features.

Cancer data

Let's apply the feature selection for classification on the cancer dataset. To make the task a bit harder, we add some non-informative noise features to the data. We expect the feature selection to be able to identify the features that are non-informative and remove them.

```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()
# get deterministic random numbers
rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
# add noise features to the data
# the first 30 features are from the dataset, the next 50 are noise
X_w_noise = np.hstack([cancer.data, noise])
X_train, X_test, y_train, y_test = train_test_split(
 X_w_noise, cancer.target, random_state=0, test_size=.5)
# use f_classif (the default) and SelectPercentile to select 50% of features:
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
# transform training set:
X_train_selected = select.transform(X_train)
print(X_train.shape)
print(X_train_selected.shape)
(284, 80)
(284, 40)
```



Post-Selection

Let's compare the performance of logistic regression on all features against the performance using only the selected features:

```
from sklearn import preprocessing
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

# transform test data:
X_test_selected = select.transform(X_test)
pipe = make_pipeline(StandardScaler(), LogisticRegression())
pipe.fit(X_train, y_train)
print("Score with all features: %f" % pipe.score(X_test, y_test))
pipe.fit(X_train_selected, y_train)
print("Score with only selected features: %f" % pipe.score(X_test_selected, y_test))
Score with all features: 0.954386
Score with only selected features: 0.964912
```

In this case, removing the noise features improved performance, even though some of the original features where lost. This was a very simple synthetic example, though, and outcomes on real data is usually mixed. Univariate feature selection can still be very helpful if there is such a large number of features that building a model on them is infeasible, or if you suspect that many features are completely uninformative.

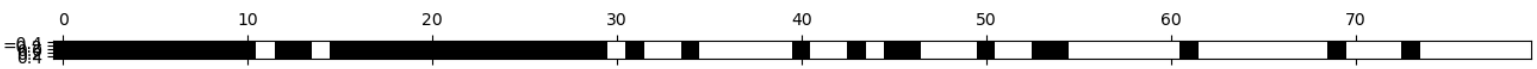
Model-based Feature Selection

The model that is used for feature selection needs to provide some measure of importance for each feature, so that they can be ranked by this measure.

- Decision trees and decision tree based models provide feature importances.
- Linear models have coefficients which can be used by considering the absolute value.
- Linear models with L1 penalty learn sparse coefficients, which only use a small subset of features. This can be viewed as a form of feature selection for the model itself, but can also be used as a preprocessing step to select features for another model.
- In contrast to univariate selection, model-based selection considers all features at once, and so can capture interactions

Cancer

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(RandomForestClassifier(n_estimators=100, random_state=42), threshold="median")
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print(X_train.shape)
print(X_train_l1.shape)
(284, 80)
(284, 40)
mask = select.get_support()
# visualize the mask. black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
Score with only selected features: 0.961404
```



Iterative Feature Selection

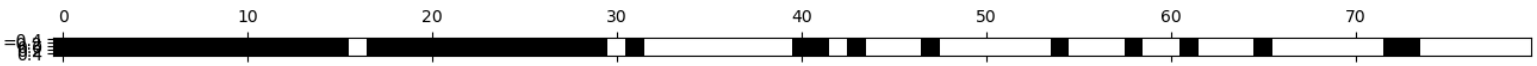
In univariate testing, we build used no model, while in model based selection we used a single model to select features. In iterative feature selection, a series of models is built, with varying numbers of features.

- Starting with no features and adding features one by one, until some stopping criterion is reached.
- Starting with all features and removing features one by one, until some stopping criterion is reached.

Because a series of models is built, these methods are much more computationally expensive then the methods we discussed above. 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 pre-specified number of features is 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.

RFE

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42), n_features_to_select=40)
#select = RFE(LogisticRegression(penalty="l1"), 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')
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)
pipe = make_pipeline(StandardScaler(), LogisticRegression())
pipe.fit(X_train_rfe, y_train)
print("Score with only selected features: %f" % pipe.score(X_test_rfe, y_test))
Score with only selected features: 0.985965
```



The feature selection got better compared to the univariate and model based selection, but one feature was still missed. Running the above code takes significantly longer than the model based selection, because a random forest model is trained 40 times, once for each feature that is dropped.

Feature Selection

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 to reduce the amount of features needed, for example to speed up prediction, or 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

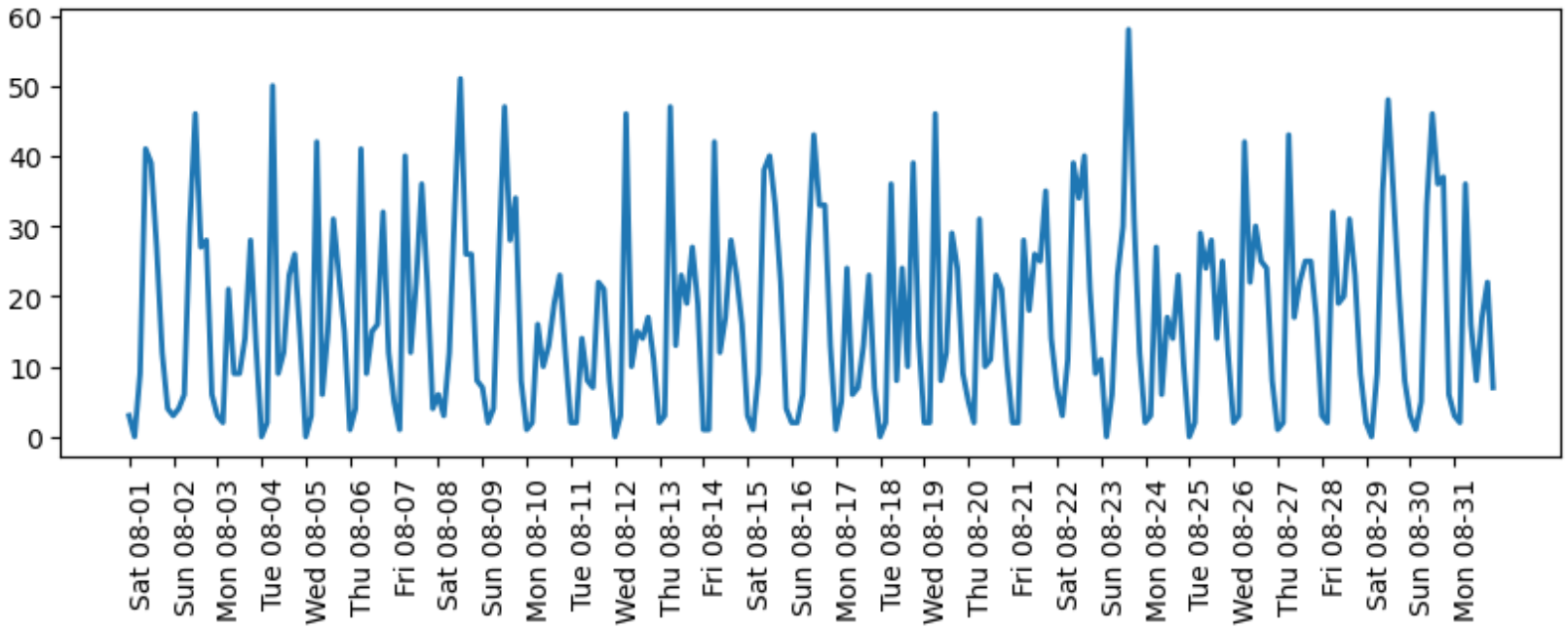
Feature engineering is often an important place to use *expert knowledge* for a particular application. While the purpose of machine learning often is to avoid having to create a set of expert-designed rules, that doesn't mean that prior knowledge of the application or domain should be discarded. Often, domain experts can help in identifying useful features that are much more informative than the initial representation of the data.
Consider the prediction of flight prices.

- Flights are usually more expensive during school holidays or around public holidays.
- While some holidays can potentially be learned from the dates, like Christmas, others might depend on the phases of the moon (like Hannukah and Easter), or be set by authorities like school holidays.
- These events can not be learned from the data if each flight is only recorded using the date. 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 non-informative for flight prices, augmenting the data with this information doesn't hurt.

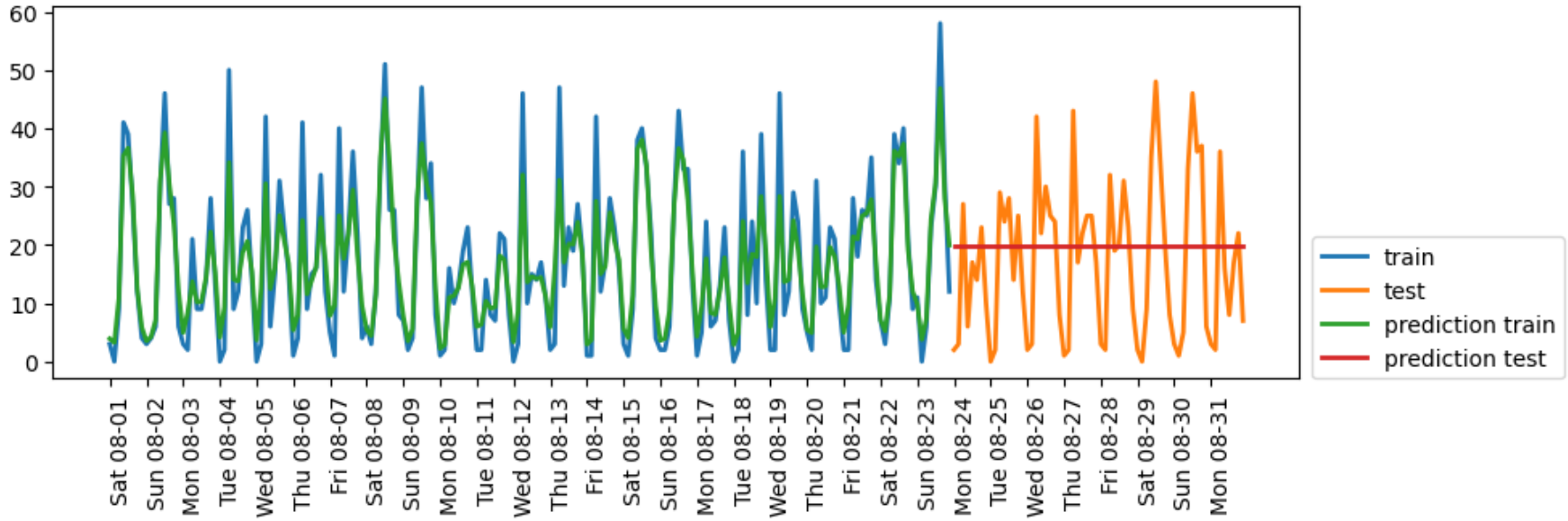
Common Sense

In New York, there is 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 anonymous 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' house - so he knows if any bikes will be left for him.



Failure of Random Forest on Time Series

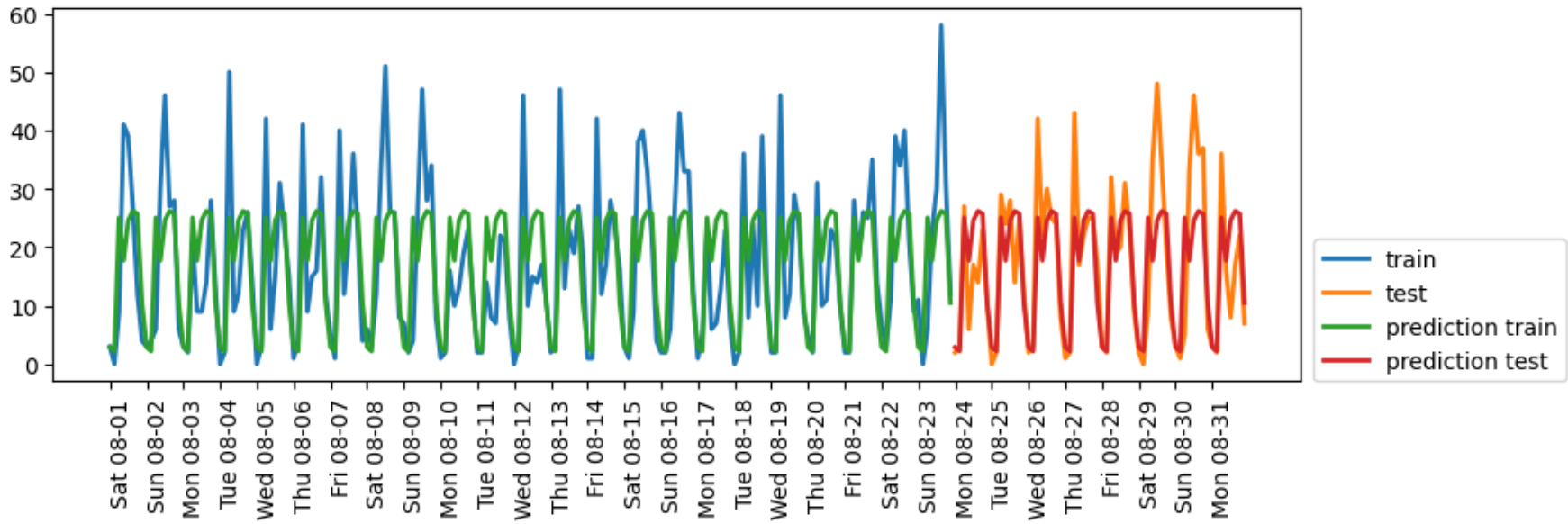
When evaluating a prediction task on a time series like this, we usually want to learn from the past and predict for the future. This means when doing a split into a training and a test set, we want to use all the data up to a certain date as training set, and all the data past that date as a test set. This is how we would usually use time series prediction: given everything that we know about rentals in the past, what do we think will happen tomorrow?



The problem lies in the combination of our feature and the random forest. The value of the POSIX time feature for the test set is outside of the range of the feature values on the training set: the points in the test set have time stamps that are later than all the points in the training set.

Expert Knowledge

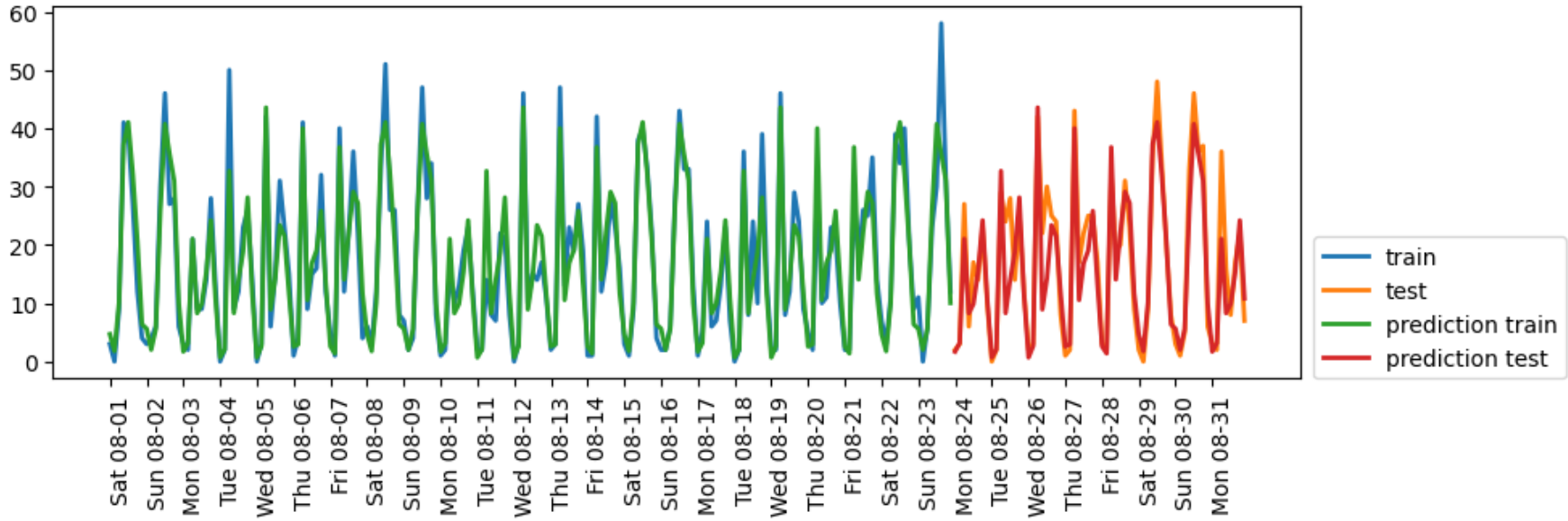
Clearly we can do better than this. This is where our "expert knowledge" comes in. From looking at the rental figures on the training data, two factors seem to be very important: the time of day, and the day of the week. So let's add these two features. We can't really learn anything from the POSIX time, so we drop that feature.



Now the predictions have the same pattern for each day of the week. The R^2 is already much better, but the predictions clearly miss the weekly pattern.

Expert Knowledge

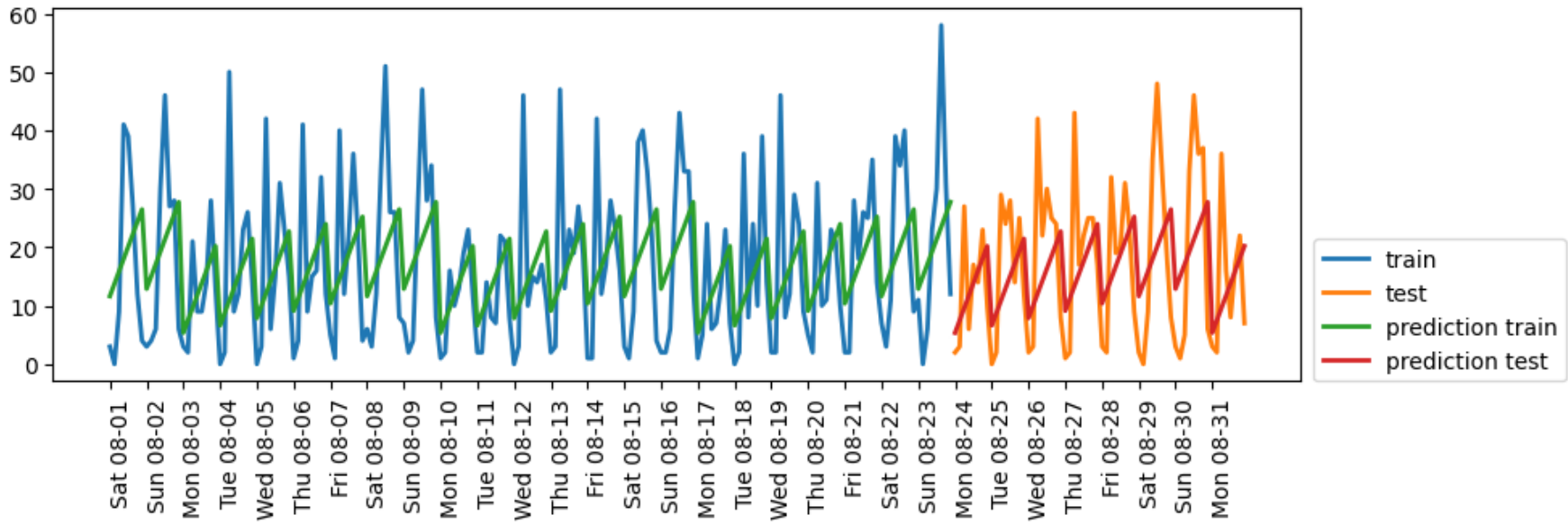
Now let's also add the day of the week:



Now, we have a model that models the periodic behavior considering day of week and time of day. It has an R^2 of 0.84, and show pretty good predictive performance.

Linear Regression

What this model likely is learning is the mean number of rentals for each combination of weekday and time of day from the first 23 days of August. This would actually not require a complex model like a random forest. So let's try with a simpler model, LinearRegression:



Linear Regression works much worse, and the periodic pattern looks odd. The reason for this is that we encoded day of the week and time of the day using integers, which are interpreted as categorical variables. However, the patterns are much more complex that, which we can capture by interpreting the integers as categorical variables, by transforming them using OneHotEncoder.

Linear Regression