OSEMN methodology
Step 4: Model
Classification methodology
Excerpts from Python Machine Learning
Second Edition
By Sebastian Raschka and Vahid Mirjalili[1]
and other sources

Stepan Oskin

September 14, 2019

Abstract

1 Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice; each algorithm has its own quirks and is based on certain assumptions. To restate the No Free Lunch theorem by David H. Wolpert[2], no single classifier works best across all possible scenarios. In practice, it is always recommended that you compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem; these may differ in the number of features or samples, the amount of noise in a dataset, and whether the classes are linearly separable or not.

2 Five main steps to train a machine learning algorithm

Eventually, the performance of a classifier—computational performance as well as predictive power—depends heavily on the underlying data that is available for learning. The five main steps that are involved in training a machine learning algorithm can be summarized as follows:

1. Selecting features and collecting training samples.
2. Choosing a performance metric.
3. Choosing a classifier and optimization algorithm.
4. Evaluating the performance of the model.
5. Tuning the algorithm.

The `scikit-learn` library in Python offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models.

3 Handling categorical data

3.1 Nominal and ordinal features

When we are talking about categorical data, we have to further distinguish between nominal and ordinal features. Ordinal features can be understood as categorical values that can be sorted or ordered. For example, t-shirt size would be an ordinal feature, because we can define an order $XL > L > M$. In contrast, nominal features don't imply any order and, to continue with the previous example, we could think of t-shirt color as a nominal feature since it typically doesn't make sense to say that, for example, red is larger than blue.

3.2 Mapping ordinal features

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the

correct order of the labels of our `size` feature, so we have to define the mapping manually. In the following simple example, let's assume that we know the numerical difference between features, for example, $XL = L+1 = M+2$:

```
size_mapping = {'XL': 3, 'L': 2, 'M': 1}
df['size'] = df['size'].map(size_mapping)
```

If we want to transform the integer values back to the original string representation at a later stage, we can simply define a reverse-mapping dictionary `inv_size_mapping = {v: k for k, v in size_mapping.items()}` that can then be used via the `pandas` `map` method on the transformed feature column, similar to the `size_mapping` dictionary that we used previously. We can use it as follows:

```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
df['size'].map(inv_size_mapping)
```

3.3 Performing one-hot encoding on nominal features

In the previous section, we used a simple dictionary-mapping approach to convert the ordinal size feature into integers. Since `scikit-learn`s estimators for classification treat class labels as categorical data that does not imply any order (nominal), we can use the convenient `LabelEncoder` to encode the string labels into integers. It may appear that we could use a similar approach to transform the nominal color column of our dataset.

However, if we stop at this point and feed the array to our classifier, we will make **one of the most common mistakes in dealing with categorical data**. Although the color values don't come in any particular order, a learning algorithm will now assume that green is larger than blue, and red is larger than green. Although this assumption is incorrect, the algorithm could still produce useful results. However, those results would not be optimal.

A common workaround for this problem is to use a technique called one-hot encoding. The idea behind this approach is to create a new dummy feature for each unique value in the nominal feature column. Here, we would convert the color feature into three new features: blue, green, and red. Binary values can then be used to indicate the particular color of a sample; for example, a blue sample can be encoded as blue=1, green=0, red=0. To perform this transformation, we can use the `OneHotEncoder` that is implemented in the `scikit-learn.preprocessing` module:

```
ohe = OneHotEncoder(categorical_features=[0])
ohe.fit_transform(X).toarray()
```

When we initialized the `OneHotEncoder`, we defined the column position of the variable that we want to transform via the `categorical_features` parameter (note that color is the first column in the feature matrix X). By default, the `OneHotEncoder` returns a sparse matrix when we use the transform method, and we converted the sparse matrix representation into a regular (dense) `NumPy` array for the purpose of visualization via the `toarray` method. Sparse matrices are a more efficient way of storing large datasets and one that is supported by many `scikit-learn` functions, which is especially useful if an array contains a lot of zeros. To omit the toarray step, we could alternatively initialize the encoder as `OneHotEncoder(..., sparse=False)` to return a regular `NumPy` array.

An even more convenient way to create those dummy features via one-hot encoding is to use the `get_dummies` method implemented in `pandas`. Applied to a DataFrame, the `get_dummies` method will only convert string columns and leave all other columns unchanged:

```
pd.get_dummies(df[['price', 'color', 'size']])
```

When we are using one-hot encoding datasets, we have to keep in mind that it introduces multicollinearity, which can be an issue for certain methods (for instance, methods that require matrix inversion). If features are highly correlated, matrices are computationally difficult to invert, which can lead to numerically unstable estimates. To reduce the correlation among variables, we can simply remove one feature column from the one-hot encoded array. Note that we do not lose any important information by removing a feature column, though; for example, if we remove the column `color_blue`, the feature information is still preserved since if we observe `color_green=0` and `color_red=0`, it implies that the observation must be blue.

If we use the `get_dummies` function, we can drop the first column by passing a True argument to the `drop_first` parameter.

The `OneHotEncoder` does not have a parameter for column removal, but we can simply slice the one-hot encoded `NumPy` array as shown in the following code snippet:

```
ohe = OneHotEncoder(categorical_features=[0])
ohe.fit_transform(X).toarray()[:, 1:]
```

4 Encoding target variable

Many machine learning libraries require that class labels are encoded as integer values. Although most estimators for classification in `scikit-learn` convert class labels to integers internally, it is considered good practice to provide class labels as integer arrays to avoid technical glitches. Using integer labels is also a recommended approach to improve computational performance due to a smaller memory footprint. To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed previously. We need to remember that class labels are not ordinal, and it doesn't matter which integer number we assign to a particular string label. Thus, we can simply enumerate the class labels, starting at 0:

```
class_mapping =
  {label:idx for idx,label in enumerate(np.unique(df['classlabel']))}
```

Next, we can use the mapping dictionary to transform the class labels into integers:

```
df['classlabel'] = df['classlabel'].map(class_mapping)
```

We can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

```
inv_class_mapping = {v:  k for k, v in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
```

Alternatively, there is a convenient `LabelEncoder` class directly implemented in scikit-learn to achieve this:

```
class_le = LabelEncoder()
y = class_le.fit_transform(df['classlabel'].values)
```

Note that the `fit_transform` method is just a shortcut for calling `fit` and `transform` separately, and we can use the `inverse_transform` method to transform the integer class labels back into their original string representation:

```
class_le.inverse_transform(y)
```

5 Multi-class classification

Most algorithms in `scikit-learn` already support multi-class classification by default via the One-versus-Rest (OvR) method. **One-versus-All (OvA)**, or sometimes also called **One-versus-Rest (OvR)**, is a technique that allows us to extend a binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the samples from all other classes are

considered negative classes. If we were to classify a new data sample, we would use our n classifiers, where n is the number of class labels, and assign the class label with the highest confidence to the particular sample. In the case of the perceptron, we would use OvA to choose the class label that is associated with the largest absolute net input value.

6 Train-test split

Comparing predictions to true labels in the test set can be understood as the unbiased performance evaluation of our model before we let it loose on the real world. To evaluate how well a trained model performs on unseen data, we will split the dataset into separate training and test datasets. Using the `train_test_split` function from `scikit-learn`'s `model_selection` module, we randomly split the X and y arrays into 30 percent test data and 70 percent training data.

Note that the `train_test_split` function already shuffles the training sets internally before splitting; otherwise, all class 0 and class 1 samples would have ended up in the training set, and the test set would consist of samples from class 2. Via the `random_state` parameter, we provided a fixed random seed (`random_state=1`) for the internal pseudo-random number generator that is used for shuffling the datasets prior to splitting. Using such a fixed `random_state` ensures that our results are reproducible.

Lastly, we took advantage of the built-in support for stratification via `stratify=y`. In this context, stratification means that the `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset.

If we are dividing a dataset into training and test datasets, we have to keep in mind that we are withholding valuable information that the learning algorithm could benefit from. Thus, we don't want to allocate too much information to the test set. However, the smaller the test set, the more inaccurate the estimation of the generalization error. Dividing a dataset into training and test sets is all about balancing this trade-off.

In practice, the most commonly used splits are 60:40, 70:30, or 80:20, depending on the size of the initial dataset. However, for large datasets, 90:10 or 99:1 splits into training and test subsets are also common and appropriate. Instead of discarding the allocated test data after model training and evaluation, it is a common practice to retrain a classifier on the entire dataset as it can improve the predictive performance of the model. While this approach is generally recommended, it could lead to worse generaliza-

tion performance if the dataset is small and the test set contains outliers, for example. Also, after refitting the model on the whole dataset, we don't have any independent data left to evaluate its performance.

References

[1] S. Raschka and V. Mirjalili, *Python Machine Learning, 2nd Ed.* . Birmingham, UK: Packt Publishing, 2 ed., 2017.

[2] D. H. Wolpert, "The Lack of A Priori Distinctions Between Learning Algorithms," *Neural Computation*, vol. 8, no. 7, pp. 1391–1420, 1996.