L6 – Representing Data and Engineering Features

- Representing your data in the right way (Very Important)
  - Categorical variables
  - Binning and discretization
  - Interactions and polynomials
  - Univariate nonlinear transformations
- Automatic feature selection
  - Univariate statistics
  - Model-based feature selection
  - Iterative feature selection
- Utilizing expert knowledge

Categorical Variables

- Task of the adult dataset: a classification task with two classes as income <=50k and >50k
  - Continuous feature: age and hours-per-week
  - Categorical feature: workclass, education, sex and occupation
- Categorical features are hard to be used in regression:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b > 0$$

  - Need to represent our data in some different way
  - A solution: *one-hot-encoding* or *one-out-of-N encoding*

workclass	Government Employee	Private Employee	Self Employed	Self Employed Incorporated
Government Employee	1	0	0	0
Private Employee	0	1	0	0
Self Employed	0	0	1	0
Self Employed Incorporated	0	0	0	1

- Can be implemented by pandas or scikit-learn
  - First, we load the data using pandas from a CSV file

```
import mglearn            import pandas as pd
import os
# The file has no headers naming the columns, so we pass header=None
# and provide the column names explicitly in "names"
adult_path = os.path.join(mglearn.datasets.DATA_PATH, "adult.data")
data = pd.read_csv(adult_path, header=None, index_col=False,
            names=['age', 'workclass', 'fnlwgt', 'education', 'education-num',
                'marital-status', 'occupation', 'relationship', 'race', 'gender',
                'capital-gain', 'capital-loss', 'hours-per-week', 'native-country', 'income'])
# For illustration purposes, we only select some of the columns
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation', 'income']]
# IPython.display allows nice output formatting within the Jupyter notebook
display(data.head())
```

  - Checking string-encoded categorical data (i.e., you need to do this for all columns in real applications)

```
print(data.gender.value_counts())
print(data.occupation.value_counts())
```

- A simple & automatic way: using get_dummies function

```
print("Original features:\n", list(data.columns), "\n")
data_dummies = pd.get_dummies(data)
print("Features after get_dummies:\n", list(data_dummies.columns))
```

  - Note that the only categorical columns will be processed
  - Continuous features age and hours-per-week were not touched
  - Categorical features were expanded into one new feature for each possible value

```
data_dummies.head()
```

  - We then separate the feature columns and the target columns

```
features = data_dummies.loc[:, 'age':'occupation_ Transport-moving']
# Extract NumPy arrays
X = features.values
y = data_dummies['income_ >50K'].values
print("X.shape: {} y.shape: {}".format(X.shape, y.shape))
```

 # Note that: the column indexing in pandas includes the end of the range (i.e., above code inclusive of 'occupation_ Transport-moving'), which is different from NumPy array (np.arrange(11)[0:10] doesn't include the entry with index 10).

- Now the data is presented in a way that scikit-learn can work with

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Test score: {:.2f}".format(logreg.score(X_test, y_test)))
```

- Categorical items encoded as numbers
  - It is not always clear whether an integer feature should be treated as continuous or discrete
    - If there is no ordering between semantics – treated as discrete
    - Otherwise, like five-star ratings – can be treated as continuous
  - To illustrate, let's create a synthetic DataFrame object

```
# create a DataFrame with an integer feature and a categorical string feature
demo_df = pd.DataFrame({'Integer Feature': [0, 1, 2, 1],
'Categorical Feature': ['socks', 'fox', 'socks', 'box']})
display(demo_df)
```

- Using get_dummies will only encode the string features but not the integer feature

```
display(pd.get_dummies(demo_df))
```

- But you can explicitly list the columns you want to encode using the columns parameter

```
demo_df['Integer Feature'] = demo_df['Integer Feature'].astype(str)
display(pd.get_dummies(demo_df, columns=['Integer Feature', 'Categorical Feature']))
```

- **scikit-learn**: Categorical variables are processed differently
  - Simple way OneHotEncoder class (applied to all input columns)

```
from sklearn.preprocessing import OneHotEncoder
# Setting sparse=false means OneHotEncoder will return a numpy array,
# not a sparse matrix
ohe = OneHotEncoder(sparse=False)
print(ohe.fit_transform(demo_df))
print(ohe.get_feature_names())
```

  - Note that: both the string and integer features were transformed
  - Better control can be realized by the ColumnTransformer class

```
display(data.head())
```

- To apply linear model to this dataset to predict income
  - Applying one-hot-encoding to the categorical variables
  - Scale the continuous variables age and hours-per-week
  - Different transformers are applied to different columns

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler

ct = ColumnTransformer(
        [("scaling", StandardScaler(), ['age', 'hours-per-week']),
        ("onehot", OneHotEncoder(sparse=False),
        ['workclass', 'education', 'gender', 'occupation']) ])
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# get all columns apart from income for the features
data_features = data.drop("income", axis=1)
```

```
# split dataframe and income
X_train, X_test, y_train, y_test = train_test_split(data_features, data.income, random_state=0)

ct.fit(X_train)
X_train_trans = ct.transform(X_train)
print(X_train_trans.shape)

# you can see that we obtained 44 features
```

- We then build a LogisticRegression model for estimation

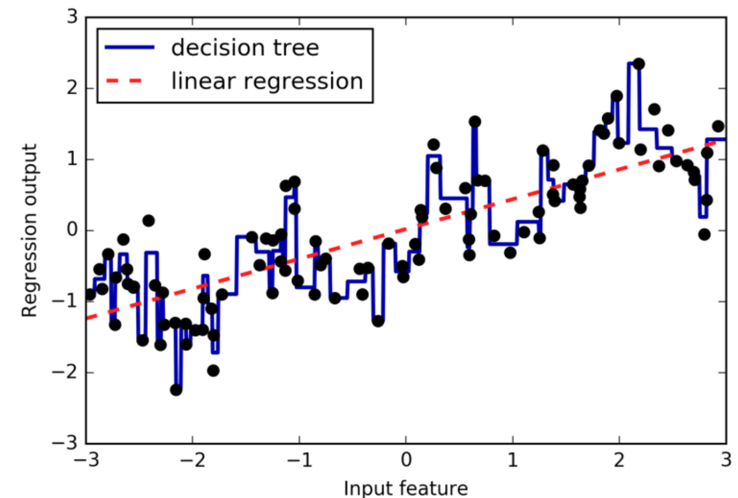
```
logreg = LogisticRegression()
logreg.fit(X_train_trans, y_train)
X_test_trans = ct.transform(X_test)
print("Test score: {:.2f}".format(logreg.score(X_test_trans, y_test)))
```

  - In this case, scaling the data did not make a difference

Binning, Discretization, Linear Models & Trees

- The best way to represent data depends not only on the semantics of the data, but also on the kind of model used
  - Linear models and tree-based models work differently with different feature representations

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
X, y = mglearn.datasets.make_wave(n_samples=100)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)

reg = DecisionTreeRegressor(min_samples_split=3).fit(X, y)
plt.plot(line, reg.predict(line), label="decision tree")
reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), label="linear regression")
plt.plot(X[:, 0], y, 'o', c='k')            plt.ylabel("Regression output")
plt.xlabel("Input feature")                 plt.legend(loc="best")
```



- One way to make linear model more powerful
  - To use binning (also called discretization) on continuous features
  - Partitioning the input range for the feature into a fixed # of bins
  - In the KBinsDiscretizer object, different strategies as
    - Uniform width (making the bin edges equidistant)
    - Quantiles of the data (having smaller bins where there is more data)

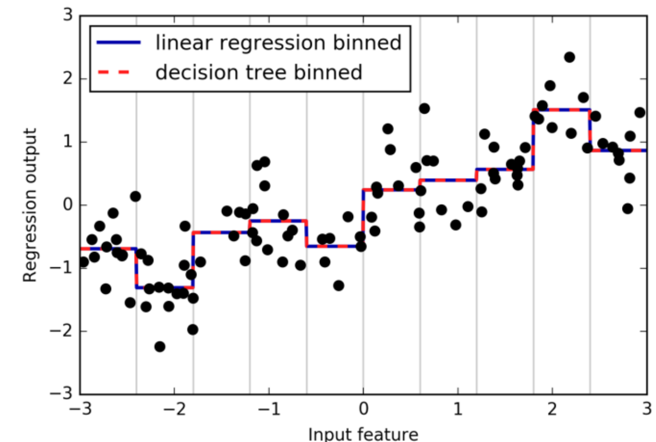
```
from sklearn.preprocessing import KBinsDiscretizer
kb = KBinsDiscretizer(n_bins=10, strategy='uniform')
kb.fit(X)
print("bin edges: \n", kb.bin_edges_)
```

  - What we did here is transform the single continuous input feature in the wave dataset into a one-hot encoded categorical feature

```
kb = KBinsDiscretizer(n_bins=10, strategy='uniform', encode='onehot-dense')
kb.fit(X)
X_binned = kb.transform(X)
```

- Now we build a new linear regression model and a new decision-tree model on the on-hot-encoded data

```
line_binned = kb.transform(line)

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='linear regression binned')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='decision tree binned')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(kb.bin_edges_[0], -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
```



- We can see that the linear model became much more flexible; because it now has a different value for each bin
- The feature representation can be further enriched by interactions and polynomials

Interactions and Polynomials

- Another way to enrich a feature representation
  - The linear model can learn not only offsets but also slopes
  - One way: adding an interaction or product feature that indicates which bin a data point is in and where it lies on the x-axis

```
X_product = np.hstack([X, X * X_binned])
print(X_product.shape)

reg = LinearRegression().fit(X_product, y)
line_product = np.hstack([line, line * line_binned])
plt.plot(line, reg.predict(line_product), label='linear regression combined')

plt.vlines(kb.bin_edges_[0], -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")      plt.ylabel("Regression output")         plt.xlabel("Input feature")
plt.plot(X[:, 0], y, 'o', c='k')
```

- Another one is to use polynomials of the original features
  - For a given feature x, we might want to consider x ** 2, x ** 3, x ** 4, and so on
  - This is implemented in the preprocessing module

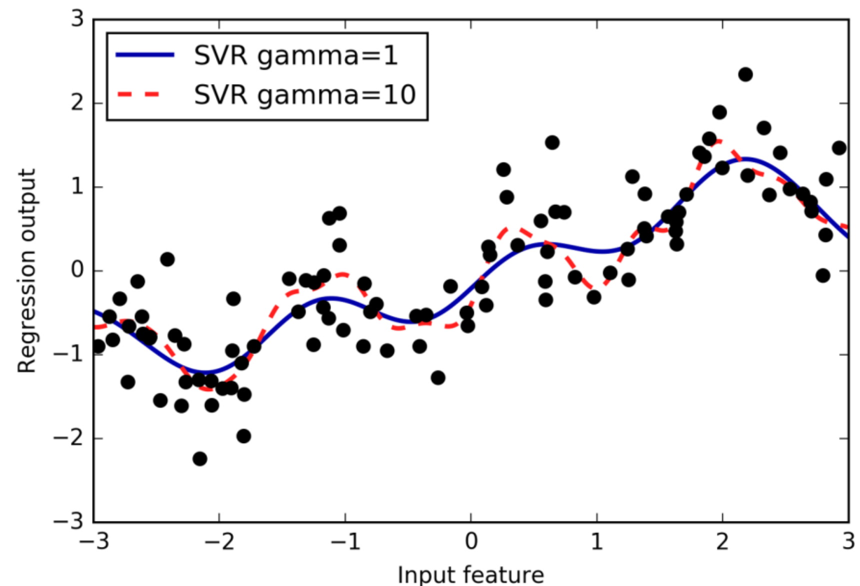
```
from sklearn.preprocessing import PolynomialFeatures
# include polynomials up to x ** 10:
# the default "include_bias=True" adds a feature that's constantly 1
poly = PolynomialFeatures(degree=10, include_bias=False)
poly.fit(X)
X_poly = poly.transform(X)

# Using a degree of 10 yields 10 features:
print("X_poly.shape: {}".format(X_poly.shape))
print("Entries of X:\n{}".format(X[:5]))
print("Entries of X_poly:\n{}".format(X_poly[:5]))

# You can obtain the semantics of the features by calling the get_feature_names
# method, which provides the exponent for each feature:
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

- Using polynomial features together with a linear regression model yields the classical model of polynomial regression:

```
reg = LinearRegression().fit(X_poly, y)
line_poly = poly.transform(line)
plt.plot(line, reg.predict(line_poly), label='polynomial linear regression')
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")        plt.xlabel("Input feature")
plt.legend(loc="best")
```

- However, polynomials of high degree tend to behave in extreme ways on the boundaries in regions with little data

```
from sklearn.svm import SVR
for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line),
                label='SVR gamma={}'.format(gamma))
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```



- Using a more complex model (e.g., a kernel SVM)
  - We are able to learn a similarly complex prediction to the polynomial regression
  - Without an explicit transformation of the features
- A more realistic application of interactions and polynomials
  - The Boston Housing dataset

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target, random_state=0)
# rescale data
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

  - Now, we extract polynomial features and interactions up to a degree of 2

```
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_poly.shape: {}".format(X_train_poly.shape))

print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

- The data originally had 13 features, which were expanded into 105 interaction features
  - Represent all possible interactions + the square of each original features + the original features + the bias
- Now compare with vs. without interactions

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(ridge.score(X_test_scaled, y_test)))
ridge = Ridge().fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(ridge.score(X_test_poly, y_test)))
```

- Different story when using a more complex model like a random forest

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100).fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(rf.score(X_test_scaled, y_test)))
rf = RandomForestRegressor(n_estimators=100).fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(rf.score(X_test_poly, y_test)))
```

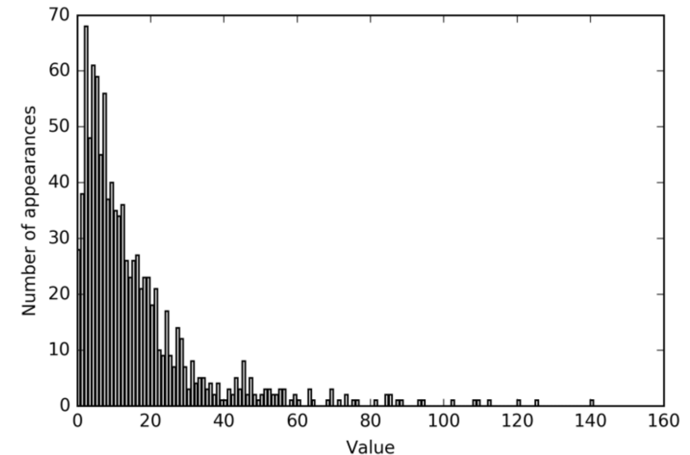
  - Even without additional features, the random forest beats the performance of Ridge
  - In summary, complicate model may perform better with original features
  - **Analysis:** polynomial features + interaction may add too much bias into the dataset

Univariate Nonlinear Transformations

- Distribution of features is important for the performance
  - Most models work best when each feature (and in regression also the target) is loosely Gaussian distributed
  - Using transformations like log and exp is a hacky but simple and efficient way to achieve this
  - A particular common case is when dealing with integer count

```
rnd = np.random.RandomState(0)
X_org = rnd.normal(size=(1000, 3))
w = rnd.normal(size=3)
X = rnd.poisson(10 * np.exp(X_org))
y = np.dot(X_org, w)

print("Number of feature appearances:\n{}".format(np.bincount(X[:, 0])))
```

```
bins = np.bincount(X[:, 0])

plt.bar(range(len(bins)), bins, color='gray')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```



- This kind of distribution, many small ones and a few very large ones, is very common in practice

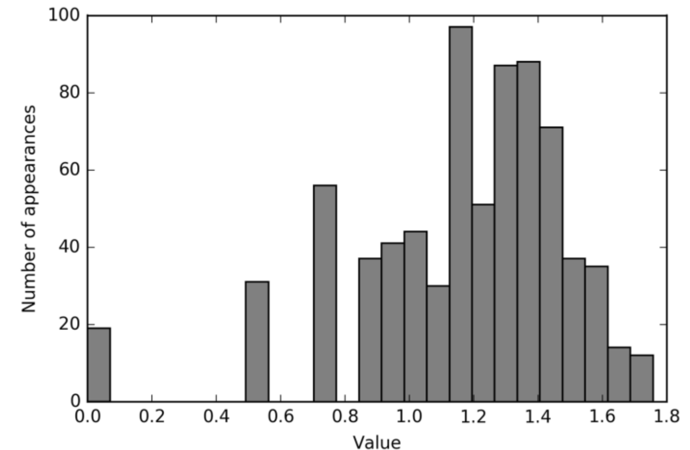
- Let's try to fit a ridge regression to this model

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
score = Ridge().fit(X_train, y_train).score(X_test, y_test)
print("Test score: {:.3f}".format(score))
```

- Applying a logarithmic transformation can help

```
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

```
plt.hist(X_train_log[:, 0], bins=25, color='gray')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```



- After transformation
  - Distribution is less asymmetric
  - Doesn’t have very large outlier any more
- Let’s build a ridge model on the new data again

```
score = Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
print("Test score: {:.3f}".format(score))
```

- The score has been significantly improved
- The transformation is irrelevant for tree-based models but might be essential for linear models
- Sometime it is also a good idea to transform the target variable y in regression

Automatic Feature Selection

- Number of features matters a lot
  - Add more features make model more complex, so that increase the chance of overfitting
  - Reduce the number of features leads to simpler models that generalize better
- Three strategies for feature selection:
  - Univariate statistics
  - Model-based selection
  - Iterative selection

*All are supervised methods

- Univariate Statistics
  - Compute whether there is statistically significant relationship between each feature and the target – i.e., analysis of variance
    - The tests consider only each feature individually
    - A feature can be discarded if it is only informative when combined with another feature
  - Use univariate feature selection in scikit-learn
    1. Choose a test
    - f_classif (the default) for classification
    - f_regression for regression
    2. Select a method to discard features based on the p-values determined
    - Discard features with too high a p-value (they are unlikely to be related to the target)
    - Method I: SelectKBest (select a fixed number k of features)
    - Method II: SelectPercentile (select a fixed percentage of features)

```python
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: {}".format(X_train.shape))
print("X_train_selected.shape: {}".format(X_train_selected.shape))
```

- We can display the selected features

```
mask = select.get_support()
print(mask)
# visualize the mask -- black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
plt.yticks(())
```

- Then we detect the performance of selected features

```
from sklearn.linear_model import LogisticRegression
# transform test data
X_test_selected = select.transform(X_test)
lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Score with all features: {:.3f}".format(lr.score(X_test, y_test)))
lr.fit(X_train_selected, y_train)
print("Score with only selected features: {:.3f}".format(lr.score(X_test_selected, y_test)))
```

- In this case, removing the noise features improve performance
- Outcome on real data are usually mixed

- Model-Based Feature Selection
  - Use a supervised machine learning model to judge the importance of each feature
  - Keep only the most important ones
  - The supervised model used for feature selection does not need to be the same model used for the final supervised learning
    - The feature_importances_ attribute of decision-tree based models
    - The coefficients of linear models0

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(
        RandomForestClassifier(n_estimators=100, random_state=42), threshold="median")
```

  - We use a random forest classifier with 100 trees - a quite complex model and much more powerful than using univariate tests

- Let’s have a look at the features that were selected

```
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_l1.shape: {}".format(X_train_l1.shape))

mask = select.get_support()
# visualize the mask -- black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
plt.yticks(())
```

- Again, we only select 40 features.
- See the performance below

```
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))
```

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

- Iterative Feature Selection
  - A series of models are built, with varying numbers of features
  - Two basic methods: incrementally 1) add or 2) remove
  - Much more computationally expensive
  - We use *recursive feature elimination* (RFE) here – remove-based
    - The model used for selection needs to provide some way to determine feature importance

```
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")
plt.yticks(())
```

- The feature selection got better results, but one feature was still missed
- Let's try the accuracy of logistic regression model below

```
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))
```

- Comparing the score of the random forest used inside RFE

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

- Reflection:
  - Model complexity vs. feature selection
  - Which is more effective?
  - Which is more efficient in computing time vs. memory?

- How about expert knowledge?

Utilizing Expert Knowledge

- Feature engineering is often an important place to use *expert knowledge* for a particular application
  - For example the case below using “common sense”
  - **Task:** predicting bicycle rentals in front of Andreas’s house
    - Citi Bike in New York operates a network of bicycle rental stations
    - For a given time and day how many people will rent a bike in front of Andreas’s house

```
citibike = mglearn.datasets.load_citibike()
print("Citi Bike data:\n{}".format(citibike.head()))
# The following example shows a visualization of the rental frequencies for the whole month
plt.figure(figsize=(10, 3))
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(), freq='D')
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")
plt.plot(citibike, linewidth=1)          plt.xlabel("Date”)          plt.ylabel("Rentals")
```

- We want to *learn from the past* and *predict for the future*
  - Input features: the date and time
    - A common way – using POSIX time (the number of seconds since January 1970 00:00:00 (aka the beginning of Unix time)
  - Output: the number of rentals in the following three hours
- We first try to use this single integer feature as our data representation

```
# extract the target values (number of rentals)
y = citibike.values
# convert to POSIX time by dividing by 10**9
X = citibike.index.astype("int64").values.reshape(-1, 1) // 10**9
```

- We then define a function to
  - split the data into training and test sets,
  - build the model and
  - visualize the result

```python
# use the first 184 data points for training, and the rest for testing
n_train = 184

# function to evaluate and plot a regressor on a given feature set
def eval_on_features(features, target, regressor):
    # split the given features into a training and a test set
    X_train, X_test = features[:n_train], features[n_train:]
    # also split the target array
    y_train, y_test = target[:n_train], target[n_train:]

    regressor.fit(X_train, y_train)
    print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)))
    y_pred = regressor.predict(X_test)                    y_pred_train = regressor.predict(X_train)

    plt.figure(figsize=(10, 3))
    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90, ha="left")
    plt.plot(range(n_train), y_train, label="train")
    plt.plot(range(n_train, len(y_test) + n_train), y_test, '-', label="test")
    plt.plot(range(n_train), y_pred_train, '--', label="prediction train")
    plt.plot(range(n_train, len(y_test) + n_train), y_pred, '--', label="prediction test")
    plt.legend(loc=(1.01, 0))              plt.xlabel("Date")            plt.ylabel("Rentals")
```

- We try the random forests as requiring very little preprocessing

```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators=100, random_state=0)
eval_on_features(X, y, regressor)
```

  - The predictions on the training set are quite good
  - However, for the test set, a constant line is predicted
- What happened?
  - A combination of our feature and the random forest
  - The value of the POSIX time feature for the test set is outside the range of the feature values in the training set
- Solution (where our “expert knowledge” comes in):
  - The time of the day and the day of the week (Two features)
  - First, let’s use only the hour of the day

```
X_hour = citibike.index.hour.values.reshape(-1, 1)
eval_on_features(X_hour, y, regressor)
```

  - Now, let’s also add the day of the week

```
X_hour_week = np.hstack([citibike.index.dayofweek.values.reshape(-1, 1), citibike.index.hour.values.reshape(-1, 1)])
eval_on_features(X_hour_week, y, regressor)
```

- In summary, we now have a model that captures the periodic behavior by considering the day of week and time of day
- Let's try to test a simpler model, LinearRegression

```
from sklearn.linear_model import LinearRegression
eval_on_features(X_hour_week, y, LinearRegression())
```

- LinearRegression works much worse, and the periodic pattern looks odd
- Reason: we encoded day of week and time of day using integers, which are interpreted as continuous variables
- Try to improve by capture this by interpreting the integers as categorical variables (i.e., using OneHotEncoder)

```
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
eval_on_features(X_hour_week_onehot, y, Ridge())
```

- This gives us a much better match than the continuous feature encoding

- Performance can be further improved by using interacted features

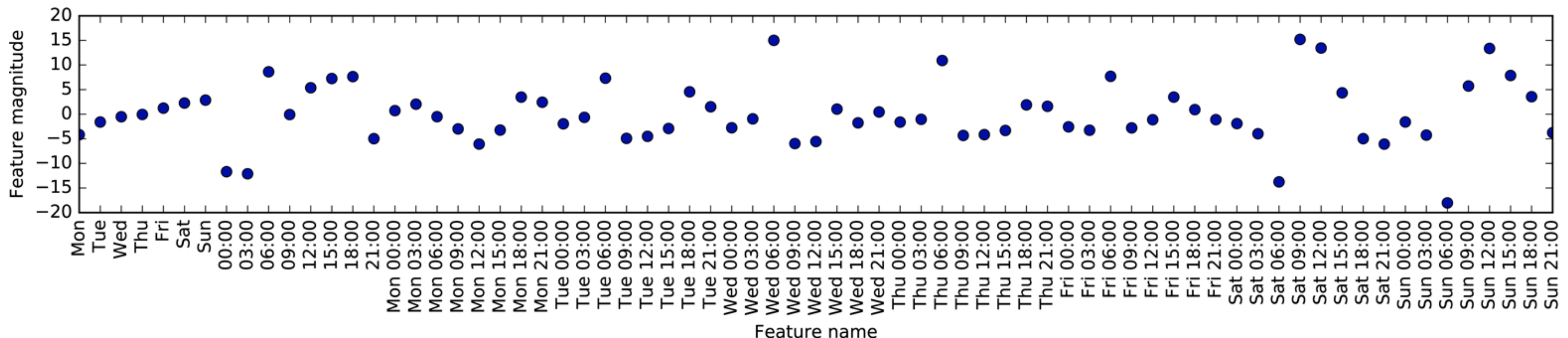
```
poly_transformer = PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

- This transformation finally yields a model that performs similarly well to the random forest
- A big benefit of this model is that: it is very clear what is learned – one coefficient for each day and time

```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features = day + hour

features_poly = poly_transformer.get_feature_names(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

- We can visualize the coefficients learned by the linear model

```
plt.figure(figsize=(15, 2))
plt.plot(coef_nonzero, 'o')
plt.xticks(np.arange(len(coef_nonzero)), features_nonzero, rotation=90)
plt.xlabel("Feature name")
plt.ylabel("Feature magnitude")
```



- In summary, important for:
  - Representing data in a way that is suitable for ML algorithm
    - E.g., one-hot-encoding categorical variables
  - Engineering new features and Utilizing expert knowledge
    - Linear model might greatly benefit from binning and adding polynomials and interactions