Machine Learning Workflow 2

Using Machine Learning Tools

Geron Chapter 2, 4.1

Last time...

- A typical workflow for supervised learning:

1. **Check and clean data**
2. Choose some candidate models based on data and task
3. Split data into **training** and **test** sets
4. Split training data into (reduced) training and **validation** sets
   - options: fixed sets, K-fold cross validation, …
5. Train candidate models on (reduced) training sets
6. Select best model based on validation set errors
7. Retrain best model on the full training set
8. Apply best model to test data
   - this gives an unbiassed estimate of the **generalisation error**

Today

- We've done an overview of the whole pipeline
  - You can train a model
  - You can use it to predict new values
  - You can measure validation/generalisation error
- Today:
  - Ways to improve on this
  - A few extra steps that can help, some practical tips
  - A closer look at a key/base model, linear regression

Missing Data

- If there are missing features:
  - can remove data, but bad if large amounts of missing data
  - can replace them with estimated/non-informative values
- sklearn supports other strategies by **imputers**
  - e.g. replace missing features with median feature value:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")

# Calculate the median for each feature
imputer.fit(training_features)

# Fill missing data (NaN) with median value
filled_features = imputer.transform(training_features)
```

Scaling Data

- Why is scaling important?
  - units are not all the same and often give very different ranges
  - often *distance* is used by the ML method, and without scaling the features with the largest values would dominate
  - practical benefits for numerical stability & default parameters
  - some ML methods *need* it to work, but not all
    - safer to use it in general as it does no harm
- Example: MinMax - scale all features to range [0, 1]

```
from sklearn.preprocessing import MinMaxScaler

# Default range is [0, 1]
scaler = MinMaxScaler()

# Find min and max value for each feature
scaler.fit(training_features)

# Apply scaling to each feature
scaled_features = scaler.transform(training_features)
```

Scaling Data

- Why is scaling important?
  - units are not all the same and often give very different ranges
  - often *distance* is used by the ML method, and without scaling the features with the largest values would dominate
  - practical benefits in terms of numerical stability & default parameters
- Example: MinMax - scale all features to range [0, 1]

```
from sklearn.preprocessing import MinMaxScaler

# Default range is [0, 1]
scaler = MinMaxScaler()

# Find min and max value for each feature
scaler.fit(training_features)

# Apply scaling to each feature
scaled_features = scaler.transform(training_features)
```

Scaling Data

- Standardised scaling:
  - scale each feature to have mean = 0, variance 1
  - there are the pros and cons of this vs MinMax
  - other options also exist (e.g. RobustScaler – uses percentiles)
    - consider whether data has: outliers, skewed distributions, multi-modal

```
from sklearn.preprocessing import StandardScaler

# Default is mean 0, variance 1
scaler = StandardScaler()

# Find mean and variance for each feature
scaler.fit(training_features)

# Apply scaling to each feature
scaled_features = scaler.transform(training_features)
```

Pipelines in sklearn

- Don’t forget to apply the **same** processing steps to your validation/test data but *without cheating!*
  - Must not use all data to determine min/max/mean/var as the validation and test data needs to be treated as ***unseen***
  - Just like training for ML, the scaling is often trained on a set and same rules apply – so in K-fold CV need to refit scaling for each fold
- We often apply the same sequence of steps to multiple datasets
  - Best practice = chain them together as a Pipeline (avoids cheating/bias)

```
from sklearn.pipeline import Pipeline

# Replace missing features with median, and scale to std distribution
std_pipeline = Pipeline([ ('imputer', SimpleImputer(strategy="median")),
                          ('std_scaler', StandardScaler()) ])

transformed_features = std_pipeline.fit_transform(training_features)
...
# Apply the same transformations to test data
trans_test_features = std_pipeline.transform(test_features)
```

Pipelines in sklearn

- Don’t forget to apply the **same** processing steps to your validation/test data but *without cheating!*
  - Must not use all data to determine min/max/mean/var as the validation and test data needs to be treated as ***unseen***
  - Just like training for ML, the scaling is often trained on a set and same rules apply – so in K-fold CV need to refit scaling for each fold
- We often apply the same sequence of steps to multiple datasets
  - **Best practice** = chain them together as a Pipeline (avoids cheating/bias)

Can build in ML method directly

```
from sklearn.pipeline import Pipeline

# Do imputation, scaling and then feed into ML method
std_pipeline = Pipeline([ ('imputer', SimpleImputer(strategy="median")),
                          ('std_scaler', StandardScaler()),
                          ('linreg', LinearRegression()) ])

std_pipeline.fit(X_train,y_train)
# Apply the same transformations (and method) to test data
y_pred = std_pipeline.predict(X_test)
```



A closer look at linear regression

- We have used linear regression as a “black box”
  - a model provided by sklearn, that can be fitted to our training features, and then predict labels for features
    - *note: all sklearn models follow this pattern…*
  - but this is a fundamental building block, so it is worth knowing more…

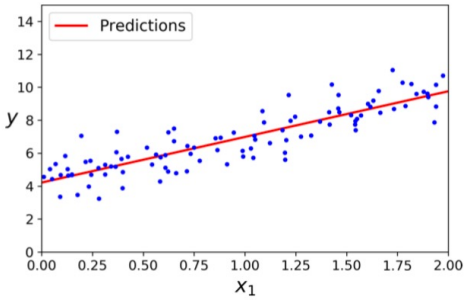
```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()

# Fit the model parameters to training data
lin_reg.fit(training_features, training_labels

# Predict labels for training features
predictions = lin_reg.predict(training_features)

# Measure prediction error, for example:
mse = mean_squared_error(training_labels, predictions)
```



Inner workings of Linear Regression

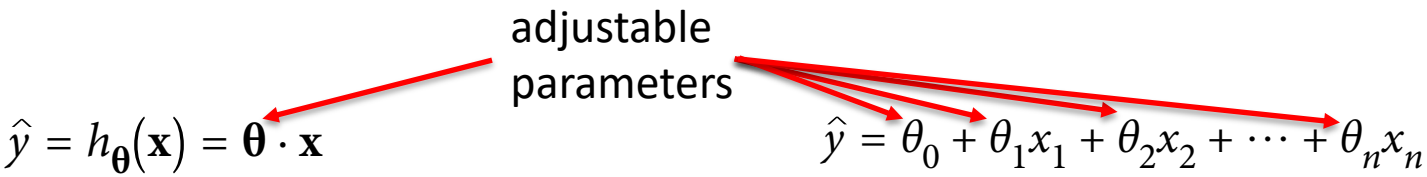
- The linear regression model:

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x} \qquad \text{or} \qquad \hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Hands-on Machine Learning, Ch 4.1

Inner workings of Linear Regression

- The linear regression model:

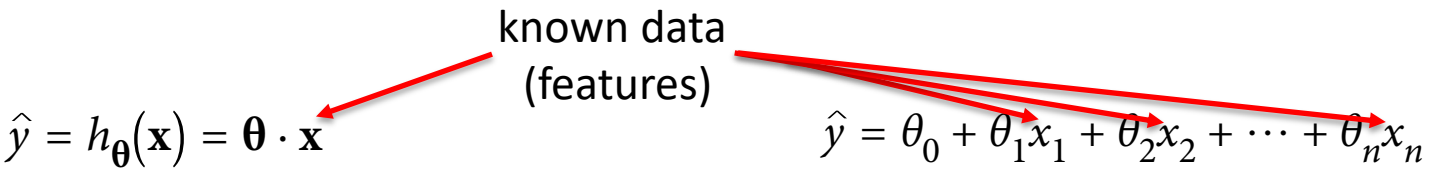


- $\boldsymbol{\theta}$ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.

Hands-on Machine Learning, Ch 4.1

Inner workings of Linear Regression

- The linear regression model:

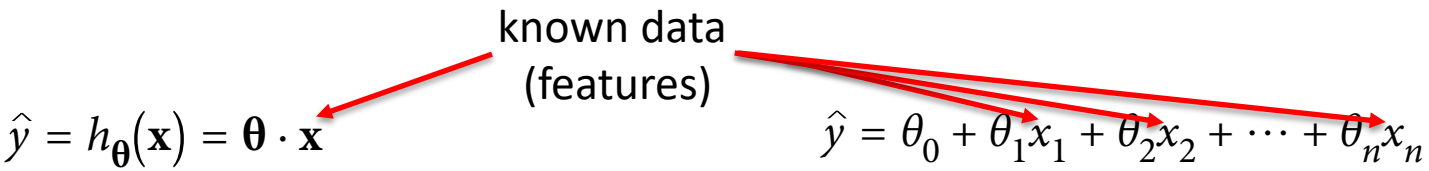


- $\boldsymbol{\theta}$ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.

Hands-on Machine Learning, Ch 4.1

Inner workings of Linear Regression

- The linear regression model:



- $\boldsymbol{\theta}$ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ is the dot product of the vectors $\boldsymbol{\theta}$ and $\mathbf{x}$, which is of course equal to $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$.
- $h_{\boldsymbol{\theta}}$ is the hypothesis function, using the model parameters $\boldsymbol{\theta}$.

Hands-on Machine Learning, Ch 4.1

Linear Regression

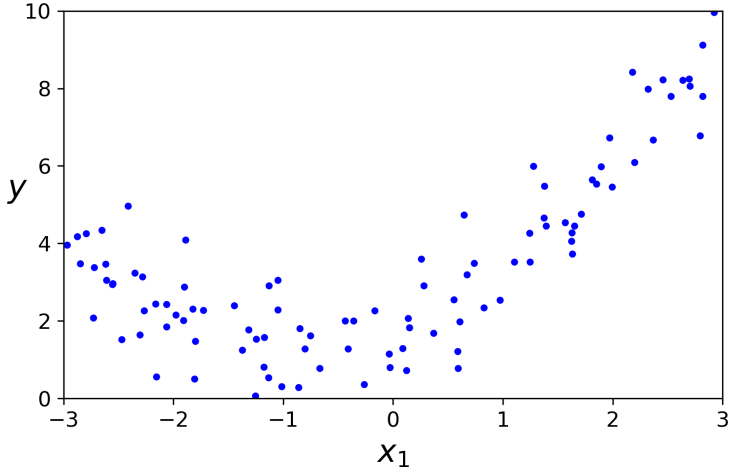
- When we fit the model, we *optimise* the parameter vector θ to *minimise* an error/loss/cost function over data points
  - e.g mean squared error (where y = target values):

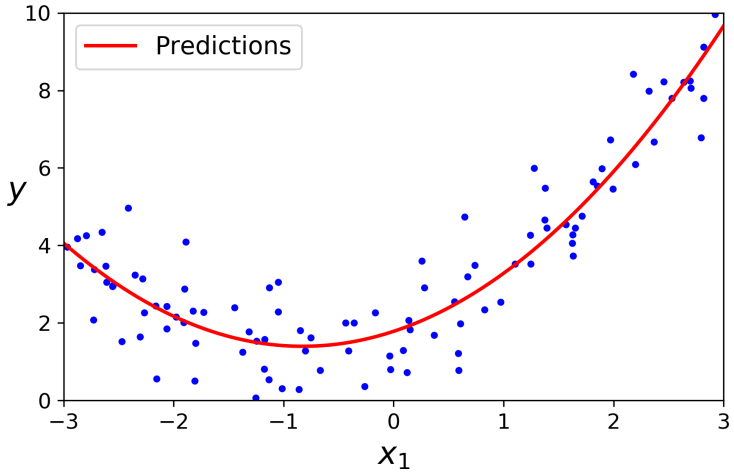
$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

- If we have *N* features, ***x*** and θ are *(N+1)*-dim vectors
  - and our linear regression model has N+1 parameters to fit
- For a linear model there is an analytic formula for the optimal result, otherwise use iterative method to find it (e.g. gradient descent)
  - analytic solutions are very rare, so most methods use some form of iterative optimizer (also for LR with large N)

Linear Regression ≠ Line Fitting

- Linear Regression can do than fit lines!
- Polynomial regression (1D shown):





- Example: fit a quadratic with 2 features

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1 x_2 + \theta_5 x_2^2$$

Linear for each θ parameter but not for data (x) values

Polynomial (linear) Regression

- Still linear in the model parameters, so same maths
  - can include non-linear functions of ***data/features***
  - sklearn includes easy ways to create polynomial features
- How to decide what degree to use?
  - quadratic, cubic, etc...

```
from sklearn.preprocessing import PolynomialFeatures

# To generate the quadratic features
poly = PolynomialFeatures(degree=2)

# Include extra quadratic features
poly_training = poly.fit_transform(training_features)
```

Parameters and Hyperparameters

- Models have ***parameters*** that are fit to data internally
- ***Hyperparameters*** are ***not changed*** during fitting
  - Manually set before fitting
  - it is up to the user/programmer to set an appropriate value
  - for example, the value of k in k-nearest neighbours
  - or the degree of a polynomial

- How to determine appropriate values?
  - if small number of options, try them all (**grid search**)
  - larger number of options: semi-randomly (but intelligently) sample from options and keep the best
  - "best" is usually measured by *validation error* (***not*** test error)

Tuning Hyperparameters

- Theory is the same as selecting between different models
- Example: grid search over values of k and weights
  - results are evaluated by 5-fold cross validation

```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsRegressor

knn_reg = KNeighborsRegressor()
param_grid = [
    # try 6 (3×2) combinations of hyperparameters
    {'n_neighbors': [3, 5, 10], 'weights': ['uniform', 'distance']},
  ]

grid_search = GridSearchCV(knn_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')

# evaluate KNN model with 6 settings
grid_search.fit(training_features, training_labels)

# what were the best settings?
grid_search.best_params_
```

Summary

- Improve the performance of your ML system by:
  - replacing data (imputation)
  - scaling data (not always *required* but better to be safe)
  - pipelines for workflows (best practice)
- A closer look at the linear regression model
  - how to use it for polynomial fitting
- Tuning hyperparameters
  - GridSearchCV
- Next time: classification models!