Recap

- **Regression** aims at predicting a **numerical target feature** based on one or multiple other (numerical) features.

- **Loss function** measures how well our model, for a specific choice of coefficients, describes the training data.

- **Ordinary least squares (OLS)** uses sum of squared errors as a loss function; closed-form optimal coefficients.

- **Gradient descent** as an optimization algorithm to find the minimum of a multivariable function.
2.3 Multiple Linear Regression

- How can we predict the value of a numerical feature $y$ based on multiple other numerical features $x_1, \ldots, x_m$?

- Again, we assume that there is a linear relationship between the target feature and the other features

\[
\hat{y} = w_0 + w_1 x_1 + \ldots + w_m x_m
\]

- Given that we now deal with multiple data points and multiple features, it is easier to formulate our optimization problem using matrices and vectors
Data Matrix

- **Data points** are encoded in a $n \times m$ data matrix $X$ with rows corresponding to **data points** and columns corresponding to **features**

\[
X = \begin{bmatrix}
  x_{1,1} & \cdots & x_{1,m} \\
  \vdots & \ddots & \vdots \\
  x_{n,1} & \cdots & x_{n,m}
\end{bmatrix}
\]

- If we want to determine an **intercept coefficient** $w_0$, then the **first column** of the data matrix is assumed to **consist of only 1s**
Target Vector and Coefficient Vector

- Values of the **target feature** provided as a $n \times 1$ **vector**

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

- Our goal is to determine an optimal $m \times 1$ **coefficient vector**

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$
Multiplying our data matrix $X$ with the coefficient vector $w$ yields a $n \times 1$ prediction vector

$$
\hat{y} = Xw = \begin{bmatrix}
w_1 x_{1,1} + \ldots + w_m x_{1,m} \\
\vdots \\
w_1 x_{n,1} + \ldots + w_m x_{n,m}
\end{bmatrix}
$$

Our loss function, capturing the sum of squared errors, can now be written as

$$
L(w) = (y - Xw)^T (y - Xw)
$$
Let’s apply this to our car dataset and try to predict fuel consumption based on horsepower and weight

```python
import pandas as pd
import numpy as np
from sklearn import linear_model

# read cars dataset
cars = pd.read_csv('/path/to/autos-mpg.data', header=None, sep='\s+)

# extract mpg values
y = cars.iloc[:,0].values

# extract horsepower and weight values
X = cars.iloc[:,[3,4]].values

# fit linear regression model
reg = linear_model.LinearRegression()
reg.fit(X,y)

# coefficients
reg.intercept_ # 45.640210840177119
reg.coef_ # [-0.04730286 -0.00579416]
```
Here, the last line computes the **mean squared error (MSE)** as another widely used measure for assessing the prediction quality of a regression model.

\[
\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2
\]
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# plot data points
for i in range(0, len(y)):
    ax.scatter(X[i, 0], X[i, 1], y[i], color='blue', marker='x')

# plot hyperplane
X0 = np.arange(min(X[:, 0]), max(X[:, 0]), 25)
X1 = np.arange(min(X[:, 1]), max(X[:, 1]), 25)
X0, X1 = np.meshgrid(X0, X1)
Z = X0.copy()
n = X0.shape[0]
m = X0.shape[1]
for i in range(0, n):
    for j in range(0, m):
        Z[i, j] = reg.predict([[X0[i, j], X1[i, j]]])
anx.plot_surface(X0, X1, Z, color='red', linewidth=0, antialiased=False)
Plotting the Hyperplane

```python
ax.set_xlabel('Power [hp]
ax.set_ylabel('Weight [lbs]
ax.set_zlabel('Fuel consumption [miles/gallon]
plt.show()
```
2.4 Handling Non-Numerical Features

- How can we make non-numerical (i.e., nominal and ordinal) features accessible to regression analysis?

- Nominal features (e.g., origin of a car) can be converted using one-hot encoding: for each value of the original feature, a binary feature is introduced, indicating whether a data point has the corresponding value for the feature.

<table>
<thead>
<tr>
<th>origin</th>
<th>origin_1</th>
<th>origin_2</th>
<th>origin_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Handling Non-Numerical Features

- **Ordinal features** (e.g., energy efficiency class) can be mapped to **integer values** preserving their order.

<table>
<thead>
<tr>
<th>energy class</th>
<th>energy class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
</tr>
</tbody>
</table>

- Note that this mapping implicitly assumes that the **differences** between **adjacent values** are **uniform**, i.e., have the same magnitude.
Predicting MPG with Origin as Additional Feature

```
import pandas as pd
import numpy as np
from sklearn import linear_model, preprocessing

# read cars dataset
cars = pd.read_csv('./path/to/autos-mpg.data', header=None, sep='\s+')

# extract mpg values
y = cars.iloc[:,0].values

# extract horsepower and weight values, apply one-hot encoding for origin
X = pd.concat([cars.iloc[:,[3,4]], pd.get_dummies(cars[7])], axis=1).values

# fit linear regression model
reg = linear_model.LinearRegression()
reg.fit(X,y)

# coefficients
reg.intercept_ # 43.974410233714622
reg.coef_ # [-0.05354417, -0.00484275, -1.2344519 , -0.27333471, 1.50778661]
```

- **Origin**: 1 corresponds to **U.S.A.**, 2 corresponds to **Europe**, and 3 corresponds to **Japan**
Origin as an additional features reduces mean squared error and allows the model to encode knowledge about fuel efficiency of cars from different origins (U.S.A. least efficient, Japan most efficient)

```python
# compute correlation coefficient	np.corrcoef(reg.predict(X),y) # 0.84810338

# compute mean squared error (MSE)
sum((reg.predict(X) - y)**2)/len(y)# 17.057355871889044
```
2.5 Polynomial Regression

- What if the relationship between our target feature $y$ and the independent features is “more complicated”?

- **Polynomial regression** allows us to estimate the optimal coefficients of a polynomial having degree $d$

$$\hat{y} = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$$

- To estimate the coefficients $w_i$, we can **precompute the values** $x^i$ and treat them just like other numerical features – no other changes are required!
import pandas as pd
import numpy as np
from sklearn import linear_model, preprocessing

# read cars dataset
cars = pd.read_csv('/path/to/autos-mpg.data', header=None, sep=' \s+ ')

# extract mpg values
y = cars.iloc[:, 0].values

# extract horsepower values
X = cars.iloc[:, [3]].values
X = X.reshape(X.size, 1)

# precompute polynomial features
poly = preprocessing.PolynomialFeatures(2)
Xp = poly.fit_transform(X)

# fit linear regression model
reg = linear_model.LinearRegression()
reg.fit(Xp, y)

# coefficients
reg.intercept_ # 56.900099702112925
reg.coef_ # [-0.46618963, 0.00123054]
Predicting MPG from Horsepower

```python
# compute correlation coefficient
np.corrcoef(reg.predict(Xp), y) # 0.82919179 (from 0.77842678)

# compute mean squared error (MSE)
sum((reg.predict(Xp) - y)**2) / len(y) # 18.984768907617223 (from 23.943662938603104)
```
import matplotlib.pyplot as plt

hp = cars.iloc[:,3].values
mpg = cars.iloc[:,0].values

hps = np.array(sorted(hp))
hps = hps.reshape(hps.size, 1)
hpsp = poly.fit_transform(hps)

plt.scatter(hp, mpg, color='blue', marker='x')
plt.plot(hps, reg.predict(hpsp), color='red', lw=2)
plt.xlabel('Power [hp]')
plt.ylabel('Fuel consumption [miles/gallon]')
plt.show()
2.6 Evaluation Fundamentals

- So far, we’ve **assessed** the **prediction quality** of our model based on the **same data** that we **used for training**.

- This is a **very bad idea** (and considered a bad foul), since:
  - we can **not accurately measure** how well our model works for **previously unseen** data points (e.g., for cars not in our dataset);
  - our model may **overfit** to the training data and loose its ability to make predictions.

- Next, we’ll see some **best practices** for **evaluating** machine learning models.
What is Overfitting?

- Let’s consider our car dataset again and learn coefficients for polynomial regression with degree $d$ based on a subset of 10 randomly chosen cars.
What is Overfitting?

- Let’s consider our car dataset again and learn coefficients for **polynomial regression** with degree $d$ based on a subset of **10 randomly chosen cars**

What is Overfitting?

- Let’s consider our car dataset again and learn coefficients for polynomial regression with degree $d$ based on a subset of 10 randomly chosen cars.

![Graph showing polynomial regression model fit to data points]

degree: 3 mse (train): 6.41892664009 mse (test): 82.0704028135
What is Overfitting?

- Let’s consider our car dataset again and learn coefficients for polynomial regression with degree $d$ based on a subset of 10 randomly chosen cars.
What is Overfitting?

- Let’s consider our car dataset again and learn coefficients for polynomial regression with degree $d$ based on a subset of 10 randomly chosen cars.
What is Overfitting?

- Let’s consider our car dataset again and learn coefficients for **polynomial regression** with degree $d$ based on a subset of **10 randomly chosen cars**
What is Overfitting?

- Let’s consider our car dataset again and learn coefficients for *polynomial regression* with degree $d$ based on a subset of 10 randomly chosen cars.

degree: 7 mse (train): 0.756138005783 mse (test): 6011912.19364
What is Overfitting?

- **Overfitting** occurs when our model describes the training data very accurately, but fails to make predictions for previously unseen data points.

- When the **number of features** is large in comparison to the **number of data points** available for training, overfitting is likely to occur.

- In that case, we learn a model that uses many features, and is thus more complex, but fails to generalize.
How to Avoid Overfitting?

- To avoid overfitting, it is good practice to assess the quality of a model based on test data that must not be used for training the model.

- The key idea is to split the available data (randomly) into training, validation, and test data.
Training-Validation-Test Splitting

- One common approach to **reliably assess the quality** of a machine learning model and **avoid overfitting** is to **randomly split** the available data into:
  - **training data** (typically about 70% of the data) is used for determining optimal coefficients.
  - **validation data** (typically about 20% of the data) is used for model selection (e.g., fixing degree of polynomial, selecting a subset of features, etc.).
  - **test data** (typically about 10% of the data) is used to measure the quality that is reported.
Another common approach, especially suitable when only limited data is available, is **k-fold cross-validation**

- **Data** is (randomly) **split** into **k folds** of **equal size**
- **k** rounds of training and validation are performed, in which
  - **(k-1) folds** serve as **training data**
  - **one fold** serves as **validation/test data**
- in the end the **mean of the quality measure** (e.g., MSE) is reported as an estimate of the overall quality
## k-Fold Cross-Validation

### Example: 5-fold cross-validation

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

- **M_1** (e.g., MSE_1)
- **M_2** (e.g., MSE_2)
- **M_3** (e.g., MSE_3)
- **M_4** (e.g., MSE_4)
- **M_5** (e.g., MSE_5)
Bias-Variance Tradeoff

- When performing **model selection** (e.g., choosing the degree for polynomial regression), we trade off between
  - **bias** as the error made due to simplifying assumptions (e.g., that there is a linear relationship)
  - **variance** as the error made due to the model being sensitive to variations in the training data
The bias/variance tradeoff

There could also be noise or problems with your measurement process. Let me show you an example. In sections 8.1 and 8.2, we played around with some two-dimensional data. This data was loaded from a file. To tell you the truth, I generated the data. The equation I used to generate the data was
\[ y = 3.0 + 1.7x + 0.1\sin(30x) + 0.06\mathcal{N}(0,1), \]
where \( \mathcal{N}(0,1) \) is a normal distribution with 0 mean and unit variance. We were trying to model this with a straight line. The best we could do with this type of model was to get the \( 3.0 + 1.7x \) part, and we'd still have an error of \( 0.1\sin(30x) + 0.06\mathcal{N}(0,1) \). We came close to this in section 8.1. In sections 8.2 and 8.3, we used locally weighted linear regression to capture the underlying structure. That structure was hard to understand, so we used different amounts of local weights to find a solution that gave us the smallest test error.

A plot of training and test error is shown in figure 8.8. The top curve is the test error, and the bottom curve is training error. If you remember from section 8.3, as we decreased the size of the kernel, our training error got smaller. This corresponds to starting on the left side of figure 8.8 and then moving to the right as the kernel becomes smaller.

![Bias-Variance Tradeoff](image)

**Source**: Harrington [1, p. 171]
2.7 Regularization

- **Regularization** (also: shrinkage) modifies the **loss function** by adding a term reflecting the **complexity of the model** (e.g., how many features have non-zero coefficients) to trade off prediction quality and model complexity.
Ridge Regression

- **Ridge regression** modifies the loss function as follows

\[
L(w) = (y - Xw)^T (y - Xw) + \lambda w^T w
\]

with

\[
w^T w = \sum_{i=1}^{m} w_i^2
\]

and \( \lambda > 0 \) as a tunable hyperparameter.
Let’s consider our car dataset again and learn coefficients for **polynomial ridge regression** with degree 5 based on a subset of **10 randomly chosen cars**.
Let’s consider our car dataset again and learn coefficients for polynomial ridge regression with degree 5 based on a subset of 10 randomly chosen cars.
Let’s consider our car dataset again and learn coefficients for polynomial ridge regression with degree 5 based on a subset of 10 randomly chosen cars.
Let’s consider our car dataset again and learn coefficients for polynomial ridge regression with degree 5 based on a subset of 10 randomly chosen cars.
import pandas as pd
import numpy as np
from sklearn import linear_model, preprocessing
import matplotlib.pyplot as plt

# read cars dataset
cars = pd.read_csv('/path/to/autos-mpg.data', header=None, sep='\s+')

# random sample of 20 cars
sample = random.sample(range(0, len(cars)), 10)
out_of_sample = list(set(range(0, len(cars))) - set(sample))

# extract mpg values for cars in sample
y = cars.iloc[sample, 0].values
y_oos = cars.iloc[out_of_sample, 0].values

# extract horsepower values for cars in sample
X = cars.iloc[sample, [3]].values
X = X.reshape(X.size, 1)

# precompute polynomial features for degree 5
poly = preprocessing.PolynomialFeatures(5)
Xp = poly.fit_transform(X)
for lmbd in [0.0, 0.0001, 0.0001, 0.001, 0.01, 0.1, 1.0, 10.0]:

    # fit linear regression model
    reg = linear_model.Ridge(alpha=lmbd, normalize=True)
    reg.fit(Xp,y)

    # plot fitted function
    hp = cars.iloc[:,3].values
    mpg = cars.iloc[:,0].values
    hps = np.array(sorted(hp))
    hps = hps.reshape(hps.size, 1)
    hpsp = poly.fit_transform(hps)

    plt.title("lambda: " + str(lmbd))
    plt.scatter(hp, mpg, color='gray', marker='x')
    plt.scatter(X, y, color='blue', marker='o')
    plt.plot(hps, reg.predict(hpsp), color='red', lw=2)
    plt.xlabel('Power [hp]')
    plt.ylabel('Fuel consumption [miles/gallon]')
    plt.xlim([min(hp), max(hp)])
    plt.ylim([min(mpg), max(mpg)])
    plt.show()
LASSO modifies the loss function as follows

$$L(w) = (y - Xw)^T (y - Xw) + \lambda \|w\|_1$$

with

$$\|w\|_1 = \sum_{i=1}^{m} |w_i|$$

and $\lambda > 0$ as a tunable hyperparameter
Summary

- **Linear regression** can be extended to predict a numerical target feature based on **multiple independent features** (matrix and vector notation is helpful here)

- **Polynomial regression** fits a polynomial of degree $k$; this can be implemented by computing additional features

- **Training, validation, and testing** have to be done on **disjoint parts** of the available data

- **Regularization** can consider model complexity
References
