lecture5-foundations-supervised-learning

October 29, 2021

1 Lecture 5: Foundations of Supervised Learning

1.0.1 Applied Machine Learning

Volodymyr Kuleshov Cornell Tech

2 Part 1: When Does Supervised Learning Work?

We have seen a number of supervised learning algorithms.

Next, let's look at why they work (and why sometimes they don't).

3 Recall: Supervised Learning

Recall our intuitive definition of supervised learning.

1. First, we collect a dataset of labeled training examples.
2. We train a model to output accurate predictions on this dataset.

4 What Is A Good Supervised Learning Model?

A good predictive model is one that makes **accurate predictions** on **new data** that it has not seen at training time.

- Accurate object detection in new scenes
- Correct translation of new sentences

Note that other definitions exist, e.g., does the model discover useful structure in the data?

5 Recall: Data Distribution

We will assume that data is sampled from a probability distribution $\mathbb{P}$, which we will call the *data distribution*. We will denote this as

$$x, y \sim \mathbb{P}.$$

The training set $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, ..., n\}$ consists of *independent and identicaly distributed* (IID) samples from $\mathbb{P}$.

6 Data Distribution: IID Sampling

The key assumption is that the training examples are *independent and identicaly distributed* (IID). * Each training example is from the same distribution. * This distribution doesn't depend on previous training examples.

Example: Flipping a coin. Each flip has same probability of heads & tails and doesn't depend on previous flips.

Counter-Example: Yearly census data. The population in each year will be close to that of the previous year.

7 Data Distribution: Example

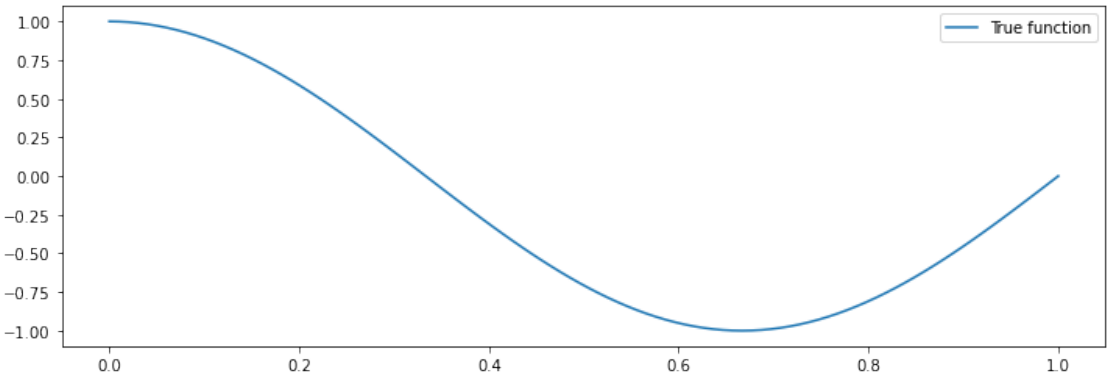
Let's implement an example of a data distribution in numpy.

[1]:
```
import numpy as np
np.random.seed(0)

def true_fn(X):
    return np.cos(1.5 * np.pi * X)
```

Let's visualize it.

[2]:
```
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

X_test = np.linspace(0, 1, 100)
plt.plot(X_test, true_fn(X_test), label="True function")
plt.legend()
```

[2]:
```
<matplotlib.legend.Legend at 0x120e92668>
```



Let's now draw samples from the distribution. We will generate random x, and then generate random y using

$$y = f(x) + \epsilon$$

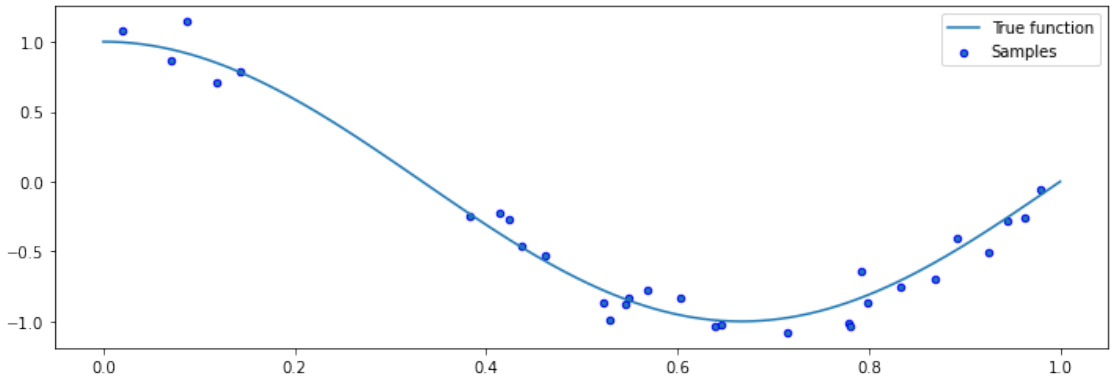
for a random noise variable ϵ.

[3]:
```
n_samples = 30

X = np.sort(np.random.rand(n_samples))
y = true_fn(X) + np.random.randn(n_samples) * 0.1
```

We can visualize the samples.

[4]:
```
plt.plot(X_test, true_fn(X_test), label="True function")
plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
plt.legend()
```

[4]:
```
<matplotlib.legend.Legend at 0x12111c860>
```



8 Data Distribution: Motivation

Why assume that the dataset is sampled from a distribution?

- The process we model may be effectively random. If y is a stock price, there is randomness in the market that cannot be captured by a deterministic model.
- There may be noise and randomness in the data collection process itself (e.g., collecting readings from an imperfect thermometer).
- We can use probability and statistics to analyze supervised learning algorithms and prove that they work.

9 Holdout Dataset: Definition

A holdout dataset

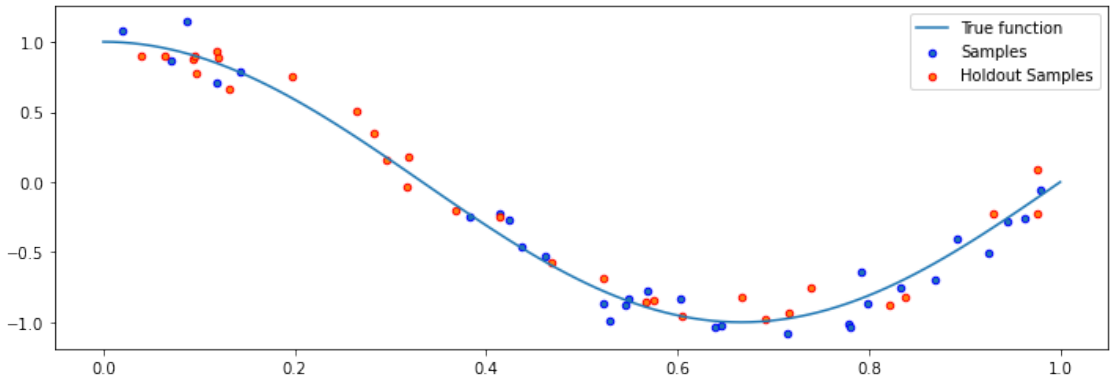
$$\dot{\mathcal{D}} = \{(\dot{x}^{(i)}, \dot{y}^{(i)}) \mid i = 1, 2, ..., m\}$$

is sampled IID from the same distribution $\mathbb{P}$, and is distinct from the training dataset $\mathcal{D}$.

Let's genenerate a holdout dataset for the example we saw earlier.

[6]:
```
n_samples, n_holdout_samples = 30, 30

X = np.sort(np.random.rand(n_samples))
y = true_fn(X) + np.random.randn(n_samples) * 0.1
X_holdout = np.sort(np.random.rand(n_holdout_samples))
y_holdout = true_fn(X_holdout) + np.random.randn(n_holdout_samples) * 0.1

plt.plot(X_test, true_fn(X_test), label="True function")
plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
plt.scatter(X_holdout, y_holdout, edgecolor='r', s=20, label="Holdout Samples")
plt.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x121440f28>
```



10 Performance on a Holdout Set

Intuitively, a supervised model f_θ is successful if it performs well on a holdout set $\dot{\mathcal{D}}$ according to some measure

$$\frac{1}{m}\sum_{i=1}^{m} L\left(\dot{y}^{(i)}, f_\theta(\dot{x}^{(i)})\right).$$

Here, $L : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is a performance metric or a loss function that we get to choose.

The choice of the performance metric L depends on the specific problem and our goals: * L can be the training objective: mean squared error, cross-entropy * In classification, L is often just accuracy: is $\dot{y}^{(i)} = f_\theta(\dot{x}^{(i)})$? * L can also implement a task-specific metric: R^2 metric (see Homework 1) for regression, F1 score for document retrieval, etc.

For example, in a classification setting, we may be interested in the accuracy of the model. Thus,

we want the % of misclassified inputs

$$\frac{1}{m}\sum_{i=1}^{m}\mathbb{I}\left(\dot{y}^{(i)} \neq f_\theta(\dot{x}^{(i)})\right)$$

to be small.

Here $\mathbb{I}(A)$ is an *indicator* function that equals one if A is true and zero otherwise.

In practice, there exist many different holdout sets $\dot{\mathcal{D}}$, and we want our model to perform well on all of them.

11 Performance on Out-of-Distribution Data

Intuitively, a supervised model f_θ is successful if it performs well in expectation on new data $\dot{x}, \dot{y}$ sampled from the data distribution $\mathbb{P}$:

$$\mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f_\theta(\dot{x})\right)\right] \text{ is "good".}$$

Here, $L : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is a performance metric and we take its expectation or average over all the possible samples $\dot{x}, \dot{y}$ from $\mathbb{P}$.

Recall that formally, an expectation $\mathbb{E}_{x)\sim P}f(x)$ is $\sum_{x\in\mathcal{X}} f(x)P(x)$ if x is discrete and $\int_{x\in\mathcal{X}} f(x)P(x)dx$ if x is continuous.

Intuitively,

$$\mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f_\theta(\dot{x})\right)\right] = \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} L\left(y, f_\theta(x)\right)\mathbb{P}(x, y)$$

is the performance on an *infinite-sized* holdout set, where we have sampled every possible point.

In practice, we cannot measure

$$\mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f_\theta(\dot{x})\right)\right]$$

on infinite data.

We approximate its performance with a sample $\dot{\mathcal{D}}$ from $\mathbb{P}$ and we measure

$$\frac{1}{m}\sum_{i=1}^{m} L\left(\dot{y}^{(i)}, f_\theta(\dot{x}^{(i)})\right).$$

If the number of IID samples m is large, this approximation holds (we call this a Monte Carlo approximation).

For example, in a classification setting, we may be interested in the accuracy of the model.

We want a small probability of making an error on a new $\dot{x}, \dot{y} \sim \mathbb{P}$:

$$\mathbb{P}\left(\dot{y} \neq f_\theta(\dot{x})\right) \text{ is small.}$$

We approximate this via the % of misclassifications on $\dot{\mathcal{D}}$ sampled from $\mathbb{P}$:

$$\frac{1}{m}\sum_{i=1}^{m}\mathbb{I}\left(\dot{y}^{(i)} \neq f_\theta(\dot{x}^{(i)})\right) \text{ is small.}$$

Here, $\mathbb{I}(A)$ is an *indicator* function that equals one if A is true and zero otherwise.

To summarize, a supervised model f_θ performs well when

$$\mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f_\theta(\dot{x})\right)\right] \text{ is "good".}$$

Under which conditions is supervised learning guaranteed to give us a good model?

12 Machine Learning Provably Works

Suppose that we choose $f \in \mathcal{M}$ on a dataset $\mathcal{D}$ of size n sampled IID from $\mathbb{P}$ by minimizing

$$\frac{1}{n}\sum_{i=1}^{n} L\left(y^{(i)}, f(x^{(i)})\right)$$

Let f^*, the best model in $\mathcal{M}$:

$$f^* = \arg\min_f \mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f(\dot{x})\right)\right]$$

Then, as $n \to \infty$, the performance of f approaches that of f^*.

13 Short Proof of Why Machine Learning Works

We say that a classification model f is accurate if its probability of making an error on a new random datapoint is small:

$$1 - \mathbb{P}\left[\dot{y} = f(\dot{x})\right] \leq \epsilon$$

for $\dot{x}, \dot{y} \sim \mathbb{P}$, for some small $\epsilon > 0$ and some definition of accuracy.

We can also say that the model f is inaccurate if it's probability of making an error on a random holdout sample is large:

$$1 - \mathbb{P}\left[\dot{y} = f(\dot{x})\right] \geq \epsilon$$

or equivalently

$$\mathbb{P}\left[\dot{y} = f(\dot{x})\right] \leq 1 - \epsilon.$$

In order to prove that supervised learning works, we will make two simplifying assumptions: 1. We define a model class $\mathcal{M}$ containing H different models

$$\mathcal{M} = \{f_1, f_2, ..., f_H\}$$

2. One of these models fits the training data perfectly (is accurate on every point) and we choose that model.

(Both of these assumptions can be relaxed.)

Claim: The probability that supervised learning will return an inaccurate model decreases exponentially with training set size n.

1. A model f is inaccurate if $\mathbb{P}[\dot{y} = f(\dot{x})] \leq 1 - \epsilon$. The probability that an inaccurate model f perfectly fits the training set is at most $\prod_{i=1}^{n} \mathbb{P}[\dot{y} = f(\dot{x})] \leq (1 - \epsilon)^n$.
2. We have H models in $\mathcal{M}$, and any of them could be in accurate. The probability that at least one of at most H inaccurate models willl fit the training set perfectly is $\leq H(1 - \epsilon)^n$.

Therefore, the claim holds.

Part 2: Evaluating Supervised Learning Models

We defined what it means for supervised learning to perform well.

Let's now look at how to evaluate its performance.

14 Datasets for Model Development

When developing machine learning models, it is customary to work with three datasets: * **Training set**: Data on which we train our algorithms. * **Development set** (validation or holdout set): Data used for tuning algorithms. * **Test set**: Data used to evaluate the final performance of the model.

15 Model Development Workflow

The typical way in which these datasets are used is: 1. **Training:** Try a new model and fit it on the training set.

2. **Model Selection**: Estimate performance on the development set using metrics. Based on results, try a new model idea in step #1.
3. **Evaluation**: Finally, estimate real-world performance on test set.

16 Validation and Test Sets

These holdout sets are used to esimate real-world performance. How should one choose the development and test set?

Distributional Consistency: The development and test sets should be from the data distribution we will see in production.

Dataset Size: Should be large enough to estimate future performance: 30% of the data on small tasks, ususally up to not more than a few thousand instances.

17 Cross-Validation

If we don't have enough data for a validation set, we can do K-fold cross-validation.

We group the data into K disjoint folds. We train the model K times, each time using a different fold for testing, and the rest for training.

Part 3: When Does Supervised Learning Not Work?

Let's now dive deeper into some failure modes of supervised learning.

18 Review: Polynomial Regression

In 1D polynomial regression, we fit a model

$$f_\theta(x) := \theta^\top \phi(x) = \sum_{j=0}^{p} \theta_j x^j$$

that is linear in θ but non-linear in x because the features

$$\phi(x) = [1 \; x \; \ldots \; x^p]$$

are non-linear. Using these features, we can fit any polynomial of degree p.

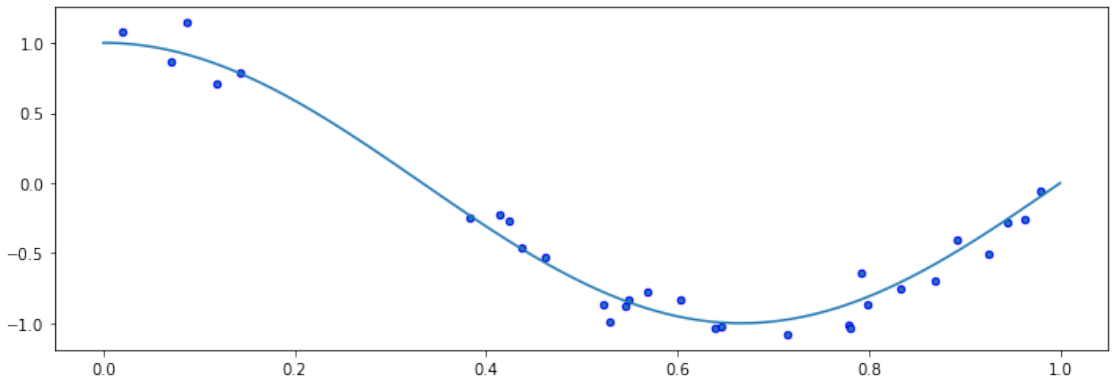
19 Polynomials Fit the Data Well

When we switch from linear models to polynomials, we can better fit the data and increase the accuracy of our models.

Consider the synthetic dataset that we have seen earlier.

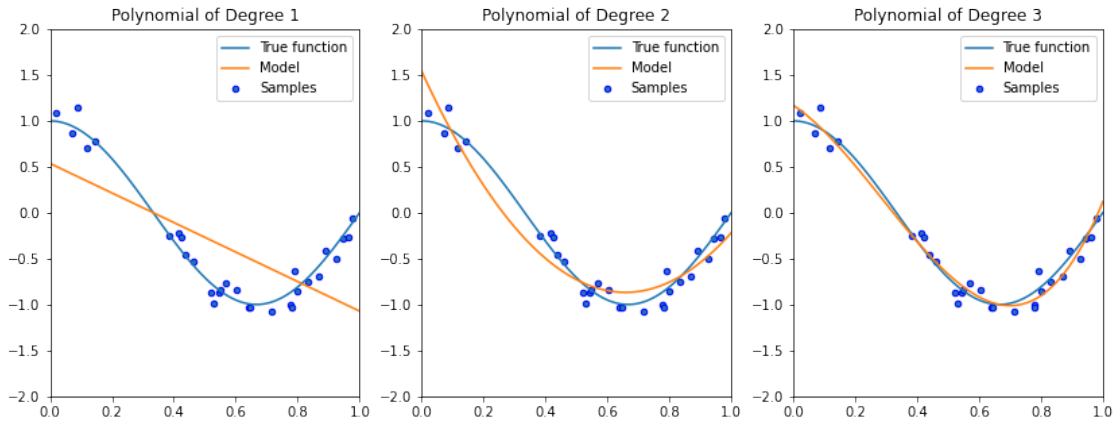
[7]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

np.random.seed(0)
n_samples = 30
X = np.sort(np.random.rand(n_samples))
y = true_fn(X) + np.random.randn(n_samples) * 0.1

X_test = np.linspace(0, 1, 100)
plt.plot(X_test, true_fn(X_test), label="True function")
plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
```

[7]:
```
<matplotlib.collections.PathCollection at 0x12e0c58d0>
```



Although fitting a linear model does not work well, qudratic or cubic polynomials improve the fit.

[8]:
```
degrees = [1, 2, 3]
plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)

    polynomial_features = PolynomialFeatures(degree=degrees[i],␣
 ↪include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("pf", polynomial_features), ("lr",␣
 ↪linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    ax.plot(X_test, true_fn(X_test), label="True function")
    ax.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    ax.scatter(X, y, edgecolor='b', s=20, label="Samples")
    ax.set_xlim((0, 1))
    ax.set_ylim((-2, 2))
    ax.legend(loc="best")
    ax.set_title("Polynomial of Degree {}".format(degrees[i]))
```

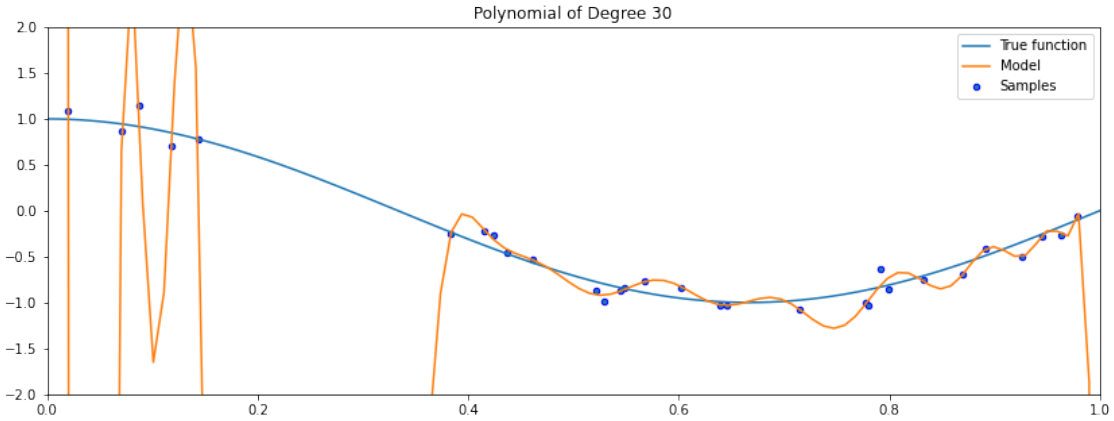


20 Towards Higher-Degree Polynomial Features?

As we increase the complexity of our model class $\mathcal{M}$ to include even higher degree polynomials, we are able to fit the data even better.

What happens if we further increase the degree of the polynomial?

[10]:
```
degrees = [30]
plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)

    polynomial_features = PolynomialFeatures(degree=degrees[i],␣
↪include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("pf", polynomial_features), ("lr",␣
↪linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    X_test = np.linspace(0, 1, 100)
    ax.plot(X_test, true_fn(X_test), label="True function")
    ax.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    ax.scatter(X, y, edgecolor='b', s=20, label="Samples")
    ax.set_xlim((0, 1))
    ax.set_ylim((-2, 2))
    ax.legend(loc="best")
    ax.set_title("Polynomial of Degree {}".format(degrees[i]))
```



As the degree of the polynomial increases to the size of the dataset, we are increasingly able to fit every point in the dataset.

However, this results in a highly irregular curve: its behavior outside the training set is wildly inaccurate.

21 Overfitting

Overfitting is one of the most common failure modes of machine learning. * A very expressive model (e.g., a high degree polynomial) fits the training dataset perfectly. * But the model makes highly incorrect predictions outside this dataset, and doesn't generalize.

22 Underfitting

A related failure mode is underfitting.

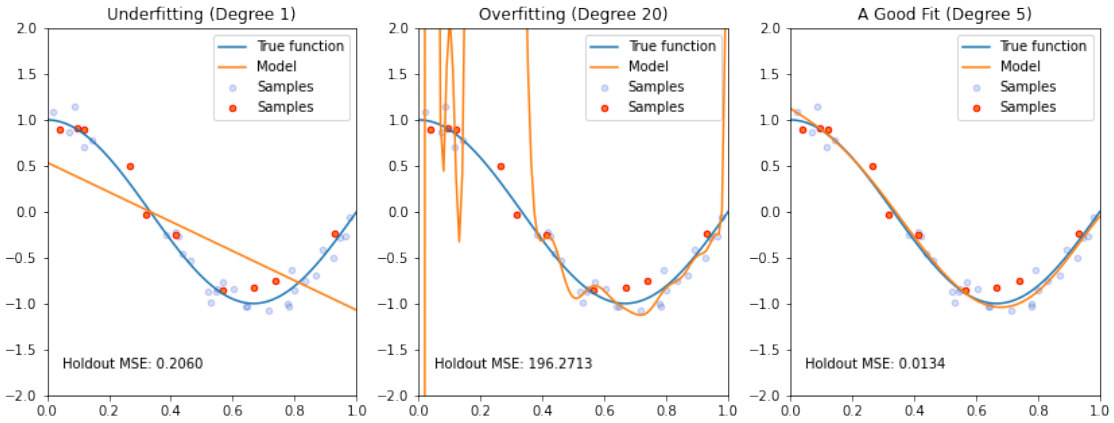
- A small model (e.g. a straight line), will not fit the training data well.
- Therefore, it will also not be accurate on a holdout set.

Finding the tradeoff between overfitting and underfitting is one of the main challenges in applying machine learning.

23 Overfitting vs. Underfitting: Evaluation

We can measure overfitting and underfitting by estimating accuracy on held out data and comparing it to the training data. * If training perforance is high but holdout performance is low, we are overfitting. * If training perforance is low but holdout performance is low, we are underfitting.

[11]:
```
degrees = [1, 20, 5]
titles = ['Underfitting', 'Overfitting', 'A Good Fit']
plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)

    polynomial_features = PolynomialFeatures(degree=degrees[i],␣
↪include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("pf", polynomial_features), ("lr",␣
↪linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    ax.plot(X_test, true_fn(X_test), label="True function")
    ax.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    ax.scatter(X, y, edgecolor='b', s=20, label="Samples", alpha=0.2)
    ax.scatter(X_holdout[::3], y_holdout[::3], edgecolor='r', s=20,␣
↪label="Samples")
    ax.set_xlim((0, 1))
    ax.set_ylim((-2, 2))
    ax.legend(loc="best")
    ax.set_title("{} (Degree {})".format(titles[i], degrees[i]))
    ax.text(0.05,-1.7, 'Holdout MSE: %.4f' % ((y_holdout-pipeline.
↪predict(X_holdout[:, np.newaxis]))**2).mean())
```



24 How to Fix Underfitting

What if our model doesn't fit the training set well? Try the following: * Create richer features that will make the dataset easier to fit. * Use a more expressive model family (neural nets vs. linear models) * Try a better optimization procedure

25 How to Fix Overfitting

We will see many ways of dealing with overftting, but here are some ideas: * Use a simpler model family (linear models vs. neural nets) * Keep the same model, but collect more training data * Modify the training process to penalize overly complex models.

Part 4: Regularization

We will now see a very important way to reduce overfitting: regularization.

26 Review: Generalization

We will assume that the dataset is governed by a probability distribution $\mathbb{P}$, which we will call the *data distribution*. We will denote this as

$$x, y \sim \mathbb{P}.$$

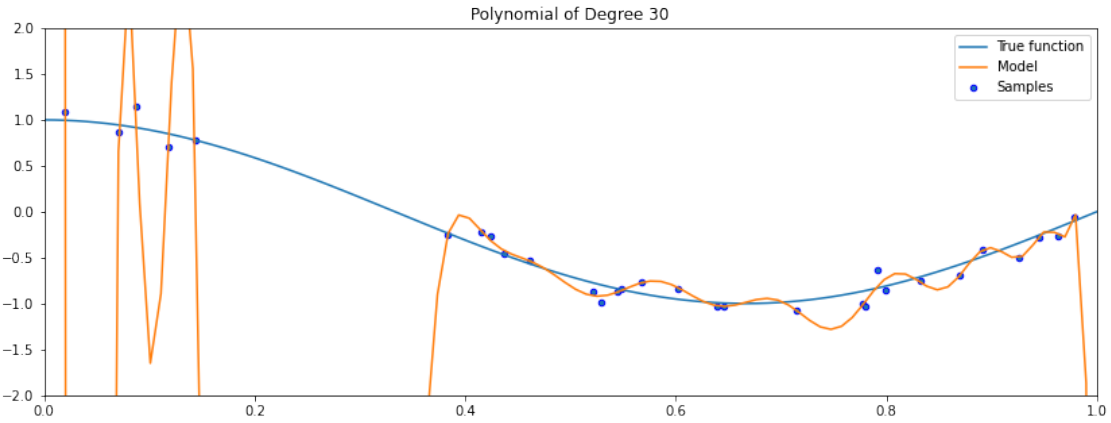
A holdout set $\dot{\mathcal{D}} = \{(\dot{x^{(i)}}, \dot{y^{(i)}}) \mid i = 1, 2, ..., n\}$ consists of *independent and identicaly distributed* (IID) samples from $\mathbb{P}$ and is distinct from the training set.

27 Review: Overfitting

Overfitting is one of the most common failure modes of machine learning. * A very expressive model (a high degree polynomial) fits the training dataset perfectly. * The model also makes wildly incorrect prediction outside this dataset, and doesn't generalize.

We can visualize overfitting by trying to fit a small dataset with a high degree polynomial.

[12]:
```
degrees = [30]
plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)

    polynomial_features = PolynomialFeatures(degree=degrees[i],␣
↪include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("pf", polynomial_features), ("lr",␣
↪linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    X_test = np.linspace(0, 1, 100)
    ax.plot(X_test, true_fn(X_test), label="True function")
    ax.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    ax.scatter(X, y, edgecolor='b', s=20, label="Samples")
    ax.set_xlim((0, 1))
    ax.set_ylim((-2, 2))
    ax.legend(loc="best")
    ax.set_title("Polynomial of Degree {}".format(degrees[i]))
```



28 Regularization: Intuition

The idea of regularization is to penalize complex models that may overfit the data.

In the previous example, a less complex would rely less on polynomial terms of high degree.

29 Regularization: Definition

The idea of regularization is to train models with an augmented objective $J : \mathcal{M} \to \mathbb{R}$ defined over a training dataset $\mathcal{D}$ of size n as

$$J(f) = \underbrace{\frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)}))}_{\text{Learning Objective}} + \underbrace{\lambda \cdot R(f)}_{\text{New Regularization Term}}$$

Let's dissect the components of this objective:

$$J(f) = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)})) + \lambda \cdot R(f).$$

- A loss function $L(y, f(x))$ such as the mean squared error.
- A regularizer $R : \mathcal{M} \to \mathbb{R}$ that penalizes models that are overly complex.
- A regularization parameter $\lambda > 0$, which controls the strength of the regularizer.

When the model f_θ is parametrized by parameters θ, we also use the following notation:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f_\theta(x^{(i)})) + \lambda \cdot R(\theta).$$

30 L2 Regularization: Definition

How can we define a regularizer $R : \mathcal{M} \to \mathbb{R}$ to control the complexity of a model $f \in \mathcal{M}$?

In the context of linear models $f_\theta(x) = \theta^\top x$, a widely used approach is L2 regularization, which defines the following objective:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, \theta^\top x^{(i)}) + \frac{\lambda}{2} \cdot ||\theta||_2^2.$$

Let's dissect the components of this objective.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, \theta^\top x^{(i)}) + \frac{\lambda}{2} \cdot ||\theta||_2^2.$$

- The regularizer $R : \Theta \to \mathbb{R}$ is the function $R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{d} \theta_j^2$. This is also known as the L2 norm of θ.
- The regularizer penalizes large parameters. This prevents us from relying on any single feature and penalizes very irregular solutions.
- L2 regularization can be used with most models (linear, neural, etc.)

31 L2 Regularization for Polynomial Regression

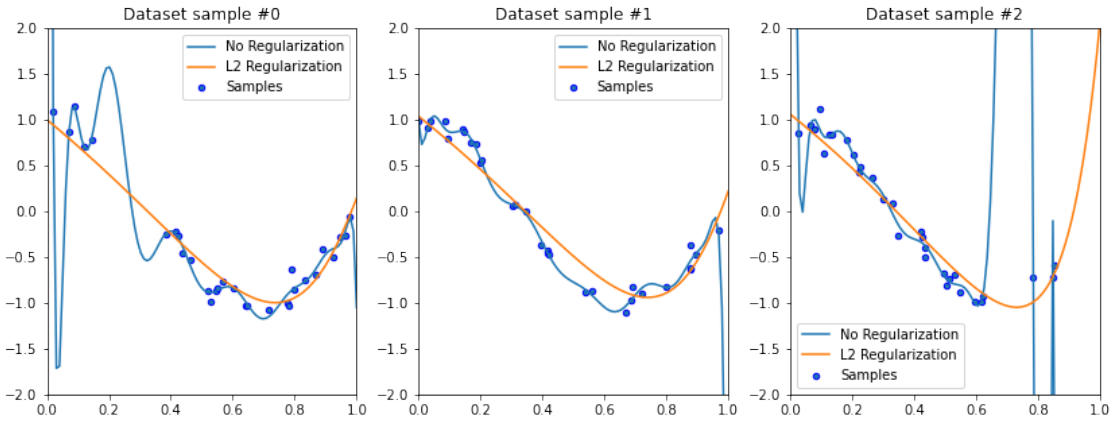
Let's consider an application to the polynomial model we have seen so far. Given polynomial features $\phi(x)$, we optimize the following objective:

$$J(\theta) = \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - \theta^\top \phi(x^{(i)})\right)^2 + \frac{\lambda}{2}\cdot||\theta||_2^2.$$

We implement regularized and polynomial regression of degree 15 on three random training sets sampled from the same distribution.

[13]:
```
from sklearn.linear_model import Ridge

degrees = [15, 15, 15]
plt.figure(figsize=(14, 5))
for idx, i in enumerate(range(len(degrees))):
    # sample a dataset
    np.random.seed(idx)
    n_samples = 30
    X = np.sort(np.random.rand(n_samples))
    y = true_fn(X) + np.random.randn(n_samples) * 0.1

    # fit a least squares model
    polynomial_features = PolynomialFeatures(degree=degrees[i],␣
↪include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("pf", polynomial_features), ("lr",␣
↪linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    # fit a Ridge model
    polynomial_features = PolynomialFeatures(degree=degrees[i],␣
↪include_bias=False)
    linear_regression = Ridge(alpha=0.1) # sklearn uses alpha instead of lambda
    pipeline2 = Pipeline([("pf", polynomial_features), ("lr",␣
↪linear_regression)])
    pipeline2.fit(X[:, np.newaxis], y)

    # visualize results
    ax = plt.subplot(1, len(degrees), i + 1)
    # ax.plot(X_test, true_fn(X_test), label="True function")
    ax.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="No␣
↪Regularization")
    ax.plot(X_test, pipeline2.predict(X_test[:, np.newaxis]), label="L2␣
↪Regularization")
    ax.scatter(X, y, edgecolor='b', s=20, label="Samples")
    ax.set_xlim((0, 1))
```

```
    ax.set_ylim((-2, 2))
    ax.legend(loc="best")
    ax.set_title("Dataset sample #{}".format(idx))
```



In order to define a very irregular function, we need very large polynomial weights.

Forcing the model to use small weights prevents it from learning irregular functions.

[14]:
```
print('Non-regularized weights of the polynomial model need to be large to fit␣
 ↪every point:')
print(pipeline.named_steps['lr'].coef_[:4])
print()

print('By regularizing the weights to be small, we force the curve to be more␣
 ↪regular:')
print(pipeline2.named_steps['lr'].coef_[:4])
```

```
Non-regularized weights of the polynomial model need to be large to fit every
point:
[-3.02370887e+03  1.16528860e+05 -2.44724185e+06  3.20288837e+07]

By regularizing the weights to be small, we force the curve to be more regular:
[-2.70114811 -1.20575056 -0.09210716  0.44301292]
```

32 How to Choose λ? Hyperparameter Search

We refer to λ as a **hyperparameter**, because it's a high-level parameter that controls other parameters.

How do we choose λ? * We select the λ with the best performance on the development set. * If we don't have enough data, we select λ by cross-validation.

33 Normal Equations for Regularized Models

How, do we fit regularized models? As in the linear case, we can do this easily by deriving generalized normal equations!

Let $L(\theta) = \frac{1}{2}(X\theta - y)^\top (X\theta - y)$ be our least squares objective. We can write the L2-regularized objective as:

$$J(\theta) = \frac{1}{2}(X\theta - y)^\top (X\theta - y) + \frac{1}{2}\lambda ||\theta||_2^2$$

This allows us to derive the gradient as follows:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \left(\frac{1}{2}(X\theta - y)^\top (X\theta - y) + \frac{1}{2}\lambda ||\theta||_2^2 \right) \\
&= \nabla_\theta \left(L(\theta) + \frac{1}{2}\lambda \theta^\top \theta \right) \\
&= \nabla_\theta L(\theta) + \lambda\theta \\
&= (X^\top X)\theta - X^\top y + \lambda\theta \\
&= (X^\top X + \lambda I)\theta - X^\top y
\end{aligned}$$

We used the derivation of the normal equations for least squares to obtain $\nabla_\theta L(\theta)$ as well as the fact that: $\nabla_x x^\top x = 2x$.

We can set the gradient to zero to obtain normal equations for the Ridge model:

$$(X^\top X + \lambda I)\theta = X^\top y.$$

Hence, the value θ^* that minimizes this objective is given by:

$$\theta^* = (X^\top X + \lambda I)^{-1} X^\top y.$$

Note that the matrix $(X^\top X + \lambda I)$ is always invertible, which addresses a problem with least squares that we saw earlier.

34 Algorithm: Ridge Regression

- **Type**: Supervised learning (regression)
- **Model family**: Linear models
- **Objective function**: L2-regularized mean squared error
- **Optimizer**: Normal equations

Part 5: L1 Regularization and Sparsity

We will now look another form of regularization, which will have an important new property called sparsity.

35 Regularization: Definition

The idea of regularization is to train models with an augmented objective $J : \mathcal{M} \to \mathbb{R}$ defined over a training dataset $\mathcal{D}$ of size n as

$$J(f) = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)})) + \lambda \cdot R(f).$$

Let's dissect the components of this objective:

$$J(f) = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)})) + \lambda \cdot R(f).$$

- A loss function $L(y, f(x))$ such as the mean squared error.
- A regularizer $R : \mathcal{M} \to \mathbb{R}$ that penalizes models that are overly complex.

36 L1 Regularizion: Definition

Another closely related approach to regularization is to penalize the size of the weights using the L1 norm.

In the context of linear models $f(x) = \theta^\top x$, L1 regularization yields the following objective:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, \theta^\top x^{(i)}) + \lambda \cdot ||\theta||_1.$$

Let's dissect the components of this objective.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, \theta^\top x^{(i)}) + \lambda \cdot ||\theta||_1.$$

- The regularizer $R : \mathcal{M} \to \mathbb{R}$ is $R(\theta) = ||\theta||_1 = \sum_{j=1}^{d} |\theta_j|$. This is known as the L1 norm of θ.
- This regularizer also penalizes large weights. It additionally forces most weights to decay to zero, as opposed to just being small.

37 Algorithm: Lasso

L1-regularized linear regression is also known as the Lasso (least absolute shrinkage and selection operator).

- **Type**: Supervised learning (regression)
- **Model family**: Linear models
- **Objective function**: L1-regularized mean squared error
- **Optimizer**: gradient descent, coordinate descent, least angle regression (LARS) and others

38 Sparsity: Definition

A vector is said to be sparse if a large fraction of its entires is zero.

L1-regularized linear regression produces *sparse parameters* θ. * This is makes the model more interpretable * It also makes it computationally more tractable in very large dimensions.

39 Regularizing via Constraints

Consider a regularized problem with a penalty term:

$$\min_{\theta \in \Theta} L(\theta) + \lambda \cdot R(\theta).$$

Alternatively, we may enforce an explicit constraint on the complexity of the model:

$$\begin{aligned} &\min_{\theta \in \Theta} L(\theta) \\ &\text{such that } \ R(\theta) \leq \lambda' \end{aligned}$$

We will not prove this, but solving this problem is equivalent to solving the penalized problem for some $\lambda > 0$ that's different from λ'.

In other words, * We can regularize by explicitly enforcing $R(\theta)$ to be less than a value instead of penalizing it. * For each value of λ, we are implicitly setting a constraint of $R(\theta)$.

40 Regularizing via Constraints: Example

This is what constraint-based regularization looks like for the linear models we have seen thus far:

$$\begin{aligned} &\min_{\theta \in \Theta} \frac{1}{2n} \sum_{i=1}^{n} \left(y^{(i)} - \theta^\top x^{(i)} \right)^2 \\ &\text{such that } \ ||\theta|| \leq \lambda' \end{aligned}$$

where $|| \cdot ||$ can either be the L1 or L2 norm.

41 L1 vs. L2 Regularization

The following image by Divakar Kapil and Hastie et al. explains the difference between the two norms.

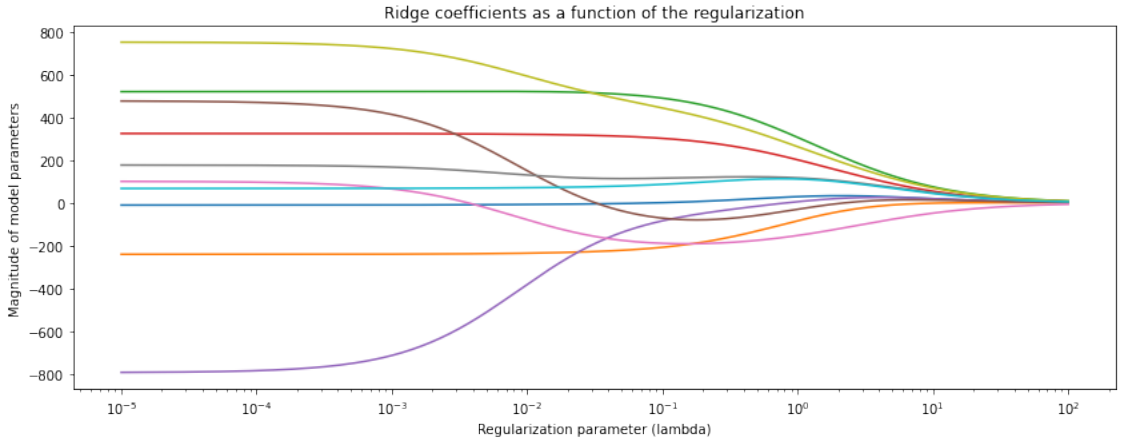
42 Sparsity: Ridge Model

To better understand sparsity, we fit Ridge and Lasso on the UCI diabetes dataset and observe the magnitude of each weight (colored lines) as a function of the regularization parameter.

Below is Ridge.

[15]:
```
# based on https://scikit-learn.org/stable/auto_examples/linear_model/
 ↪plot_ridge_path.html
from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from matplotlib import pyplot as plt

X, y = load_diabetes(return_X_y=True)

# create ridge coefficients
alphas = np.logspace(-5, 2,  )
ridge_coefs = []
for a in alphas:
    ridge = Ridge(alpha=a, fit_intercept=False)
    ridge.fit(X, y)
    ridge_coefs.append(ridge.coef_)

# plot ridge coefficients
plt.figure(figsize=(14, 5))
plt.plot(alphas, ridge_coefs)
plt.xscale('log')
plt.xlabel('Regularization parameter (lambda)')
plt.ylabel('Magnitude of model parameters')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')
```

[15]:
```
(4.466835921509635e-06,
 223.872113856834,
 -868.4051623855127,
 828.0533448059361)
```



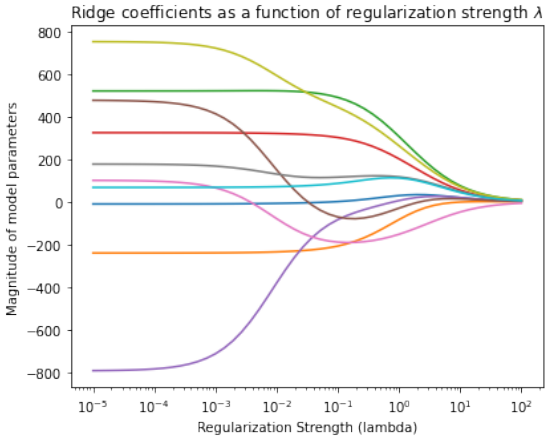
43 Sparsity: Ridge vs. Lasso

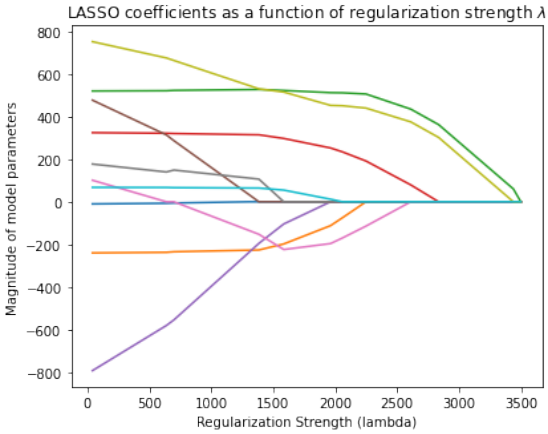
The Ridge model does not produce sparse weights. Let's now compare it to a Lasso model.

Observe how the Lasso parameters become progressively smaller, until they reach exactly zero, and then they stay at zero.

[17]:
```
# Based on: https://scikit-learn.org/stable/auto_examples/linear_model/
 ↪plot_lasso_lars.html
import warnings
warnings.filterwarnings("ignore")
from sklearn.datasets import load_diabetes
from sklearn.linear_model import lars_path

# create lasso coefficients
X, y = load_diabetes(return_X_y=True)
_, _, lasso_coefs = lars_path(X, y, method='lasso')
xx = np.sum(np.abs(lasso_coefs.T), axis=1)

# plot ridge coefficients
plt.figure(figsize=(14, 5))
plt.subplot('121')
plt.plot(alphas, ridge_coefs)
plt.xscale('log')
plt.xlabel('Regularization Strength (lambda)')
plt.ylabel('Magnitude of model parameters')
plt.title('Ridge coefficients as a function of regularization strength␣
 ↪$\lambda$')
plt.axis('tight')

# plot lasso coefficients
plt.subplot('122')
plt.plot(3500-xx, lasso_coefs.T)
ymin, ymax = plt.ylim()
plt.ylabel('Magnitude of model parameters')
plt.xlabel('Regularization Strength (lambda)')
plt.title('LASSO coefficients as a function of regularization strength␣
 ↪$\lambda$')
plt.axis('tight')
```

[17]:
```
(-133.00520290292727, 3673.000247757282, -869.357335763701, 828.4524952229654)
```


Ridge coefficients as a function of regularization strength λ
Magnitude of model parameters
Regularization Strength (lambda)


LASSO coefficients as a function of regularization strength λ
Magnitude of model parameters
Regularization Strength (lambda)