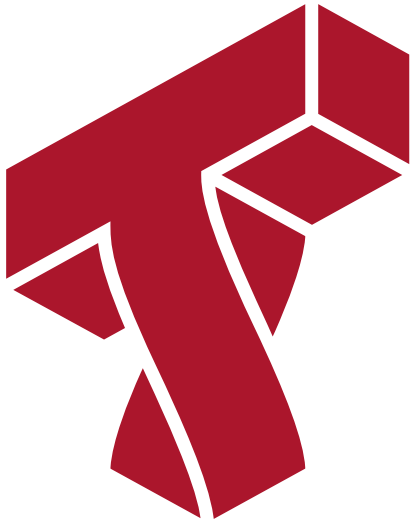


Lecture 5: Regularization

Part 1: Two Failure Modes of Supervised Learning

We have seen a number of supervised learning algorithms.

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

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.

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.

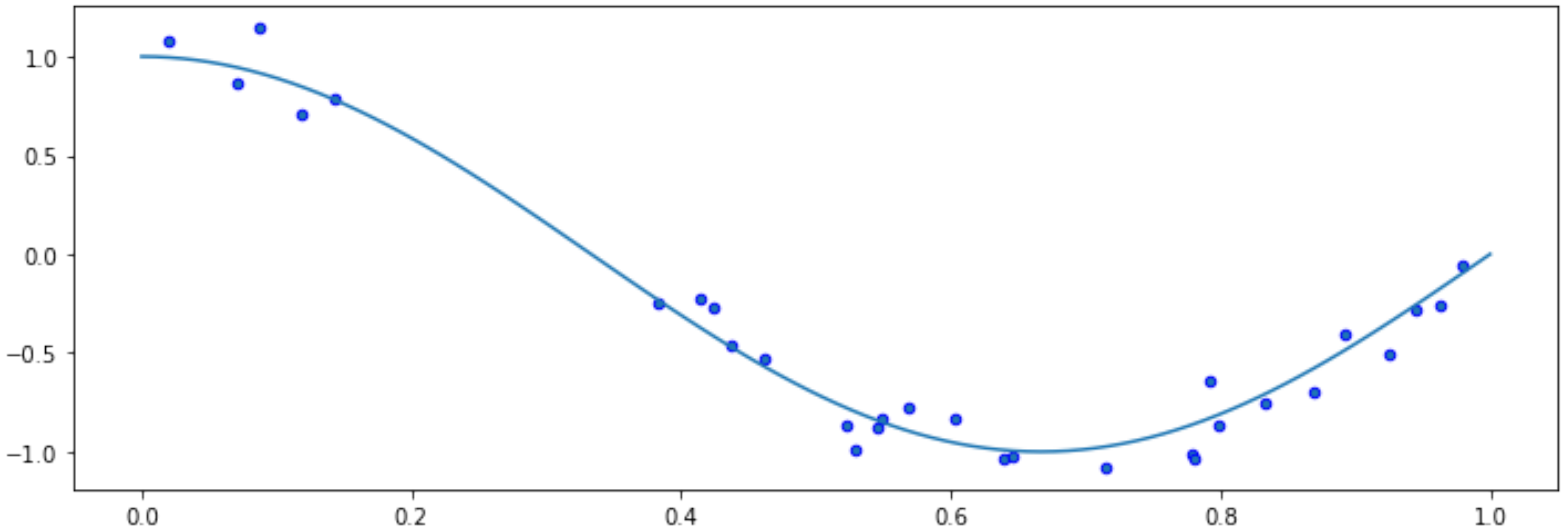
Let's generate a synthetic dataset for this demonstration.

```
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")
```

```
<matplotlib.collections.PathCollection at 0x12e0c58d0>
```

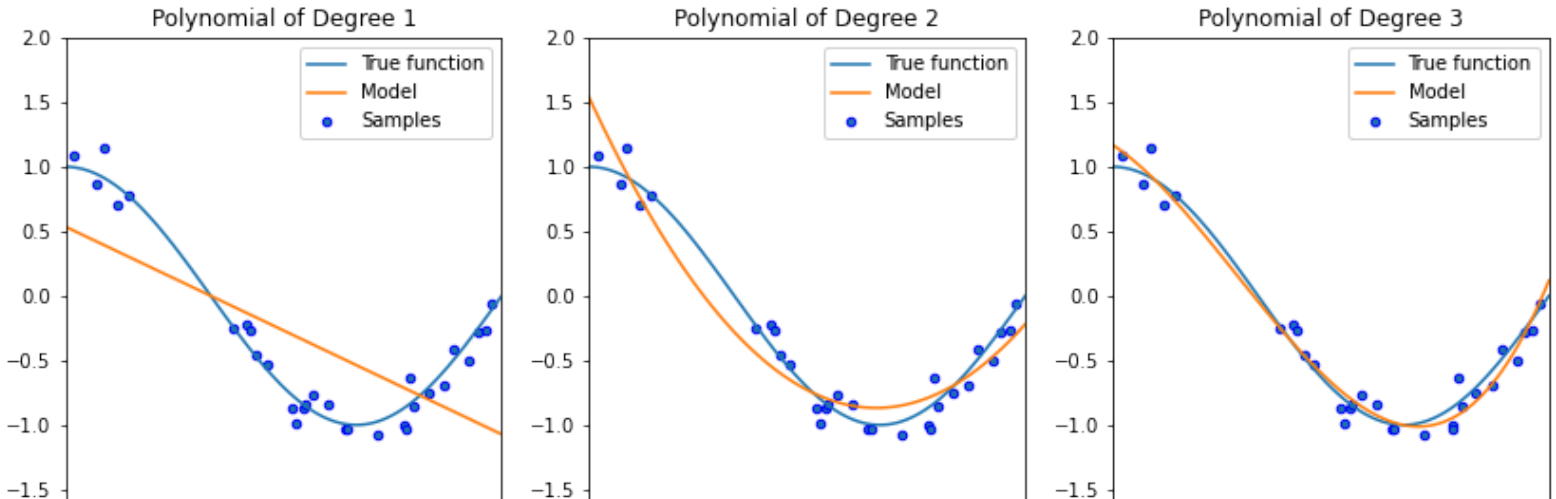


Quadratic or cubic polynomials improve the fit of a linear model.

```
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]))
```



Towards Higher-Degree Polynomial Features?

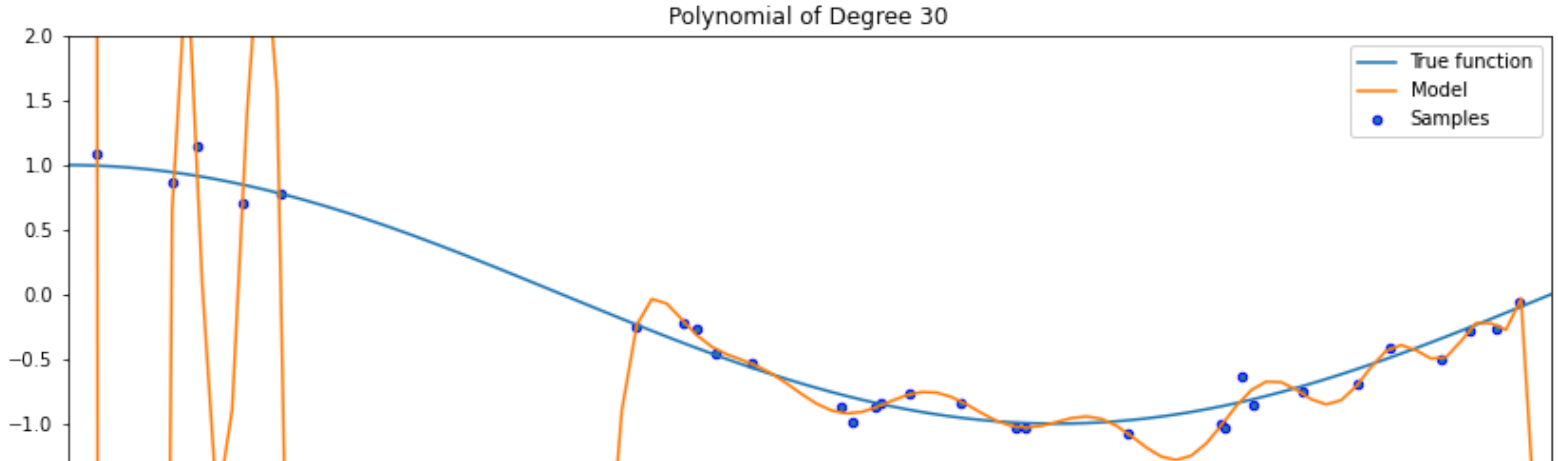
As we increase the complexity of our model class $\mathcal{M}$ to include even higher degree polynomials, we are able to fit the data even better.

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

```
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]))
```



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.

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 new data.

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

Overfitting vs. Underfitting: Evaluation

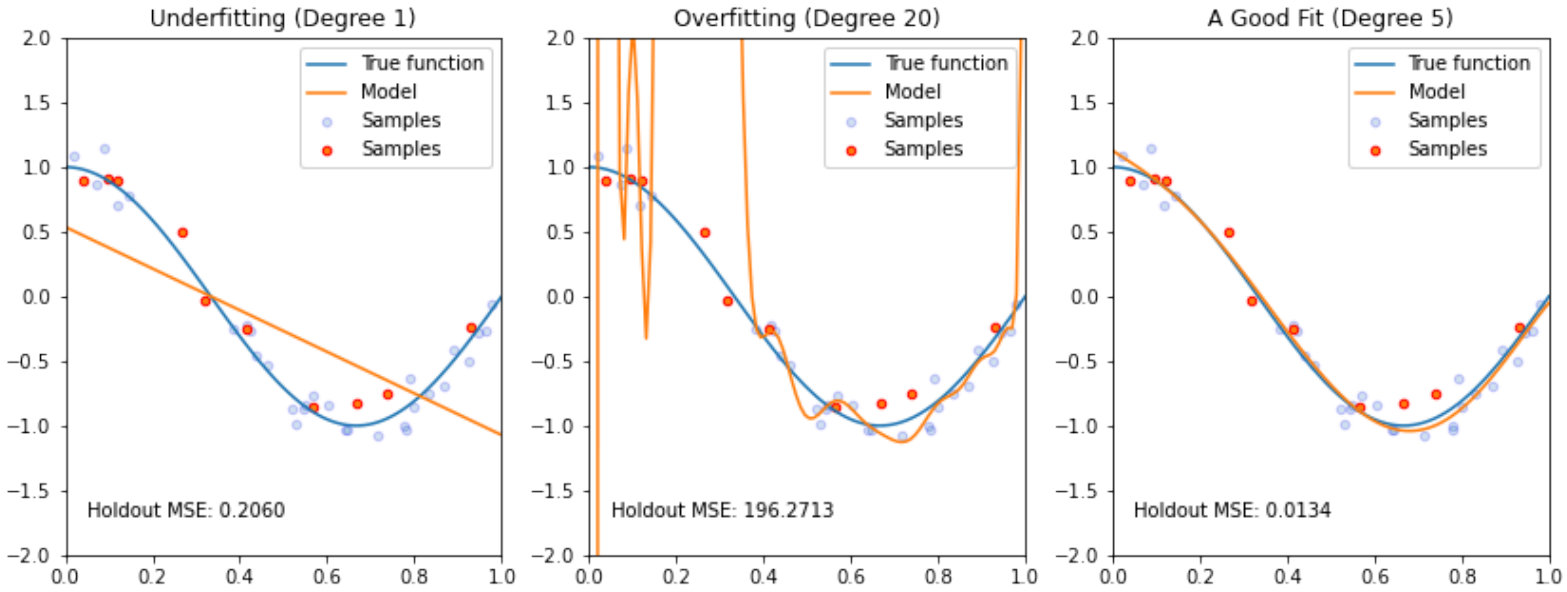
We can diagnose overfitting and underfitting by measuring performance on a separate held out dataset (not used for training).

- If training performance is **high** but holdout performance is **low**, we are overfitting.
- If training performance is **low** and holdout performance is **low**, we are underfitting.

```
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())
```



How to Fix Underfitting

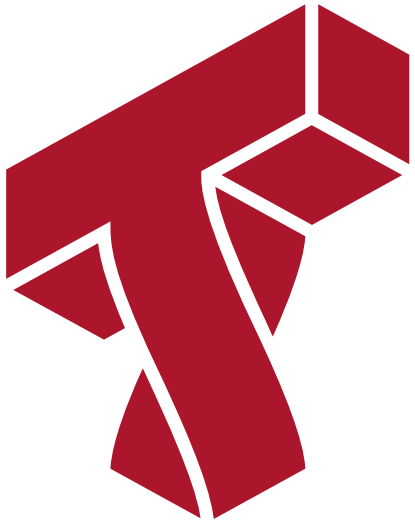
What if our model doesn't fit the training set well? Try the following:

- Create richer features that will make the dataset easier to fit.
- Use a more expressive model family (higher degree polynomials)
- Try to improve your optimization algorithm

How to Fix Overfitting

We will see many ways of dealing with overfitting, 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 2: Regularization

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

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.

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

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

- The regularizer $R : \mathcal{M} \to \mathbb{R}$ penalizes models that are complex.
- The hyperparameter $\lambda > 0$ controls the strength of the regularizer.

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.)

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.

```
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))
```

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.

```
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]
```

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:

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

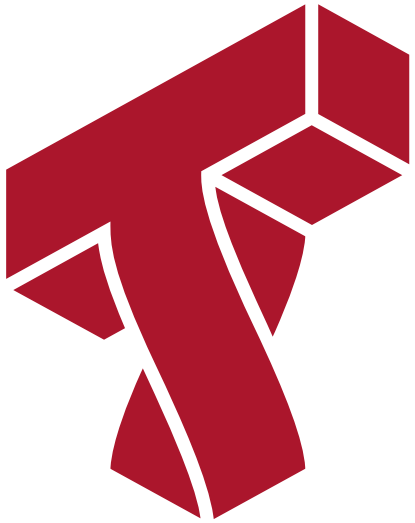
Hence, the value θ^* that minimizes this objective is given by:

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

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

Algorithm: Ridge Regression

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



Part 3: L1 Regularization and Sparsity

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

L1 Regularization: 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.

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

Sparsity: Definition

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

Sparsity: Ridge vs. Lasso

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

Below, we are going to visualize the parameters θ^* of Ridge and Lasso as a function of λ .

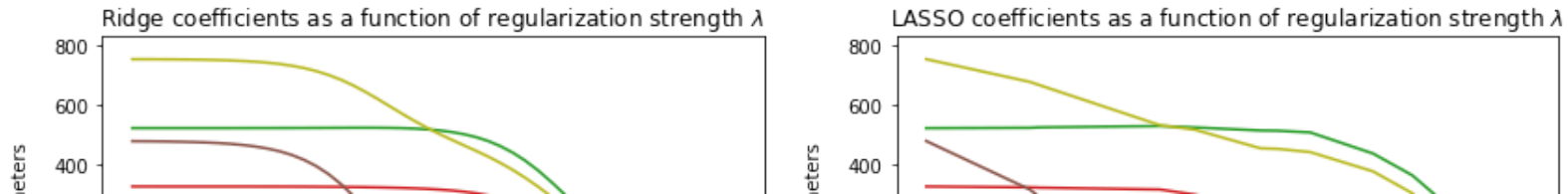
```
# Based on: https://scikit-learn.org/stable/auto_examples/linear_model/plot_lasso_lars.html
import warnings
warnings.filterwarnings("ignore")
from sklearn.datasets import load_diabetes
from sklearn.linear_model import lars_path

# 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')
```

```
(-133.00520290292727, 3673.000247757282, -869.357335763701, 828.4524952229654)
```



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'.

Regularizing via Constraints: Example

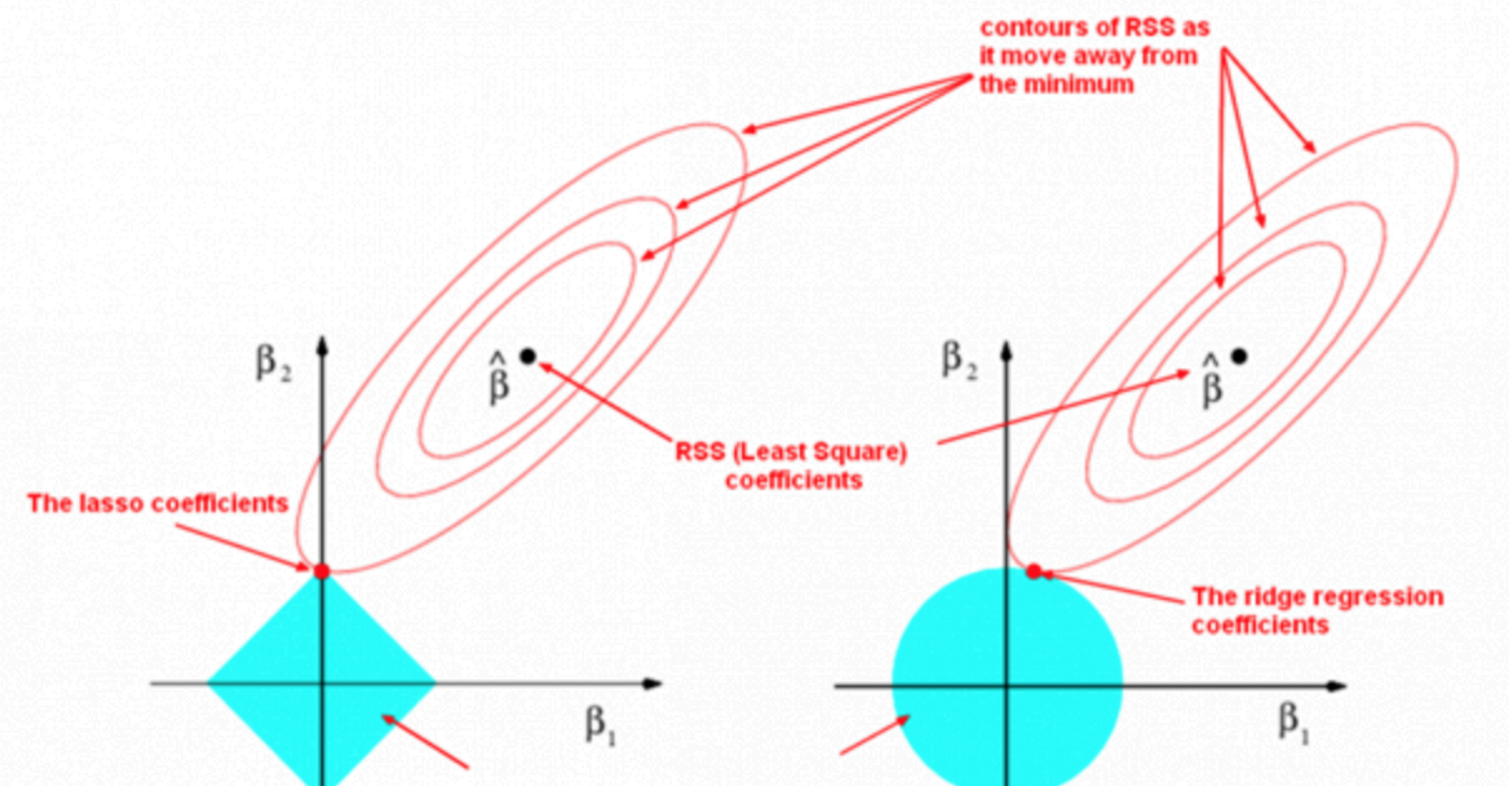
This is what constraint-based regularization looks like for the linear models we have seen thus far:

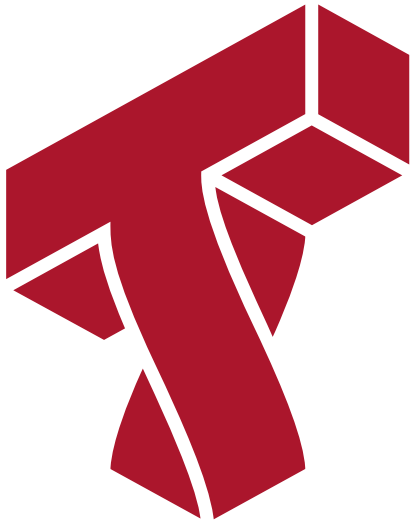
$$\min_{\theta \in \Theta} \frac{1}{2n} \sum_{i=1}^{n} \left(y^{(i)} - \theta^\top x^{(i)} \right)^2$$
$$\text{such that } ||\theta|| \leq \lambda'$$

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

L1 vs. L2 Regularization

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





Part 4: A Framework for Applying Supervised Learning

Next, we look at a framework for detecting and addressing over/underfitting.

Why Do We Need A Framework for Applying Machine Learning?

It helps to be principled. Consider the following questions:

- How do we detect overfitting or underfitting?
- How to tune the degree p in polynomial regression?
- How do we know that our model is ready to be deployed?

Our framework will provide answers that yield good models.

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?

Datasets for Model Development

When developing machine learning models, the first step is to usually split the data into three sets:

- **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.

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.

1. **Model Selection**: Estimate performance on the development set using metrics. Based on results, try a new model idea in step #1.

1. **Evaluation**: Finally, estimate real-world performance on test set.

How to Use Validation Set

There are many ways to tune a model at step #2:

- Increase/decrease degree of polynomial based on over/underfitting.
- Perform grid search to find best hyper-parameter p.
- Understand which features to add to the model.

Validation and Test Sets

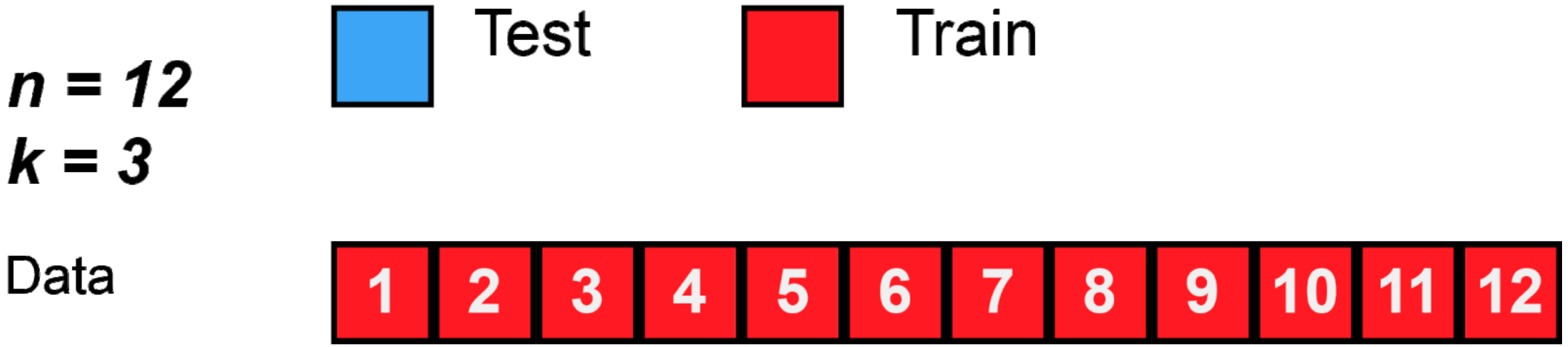
These holdout sets are used to estimate real-world performance. How should one choose the development and test set?

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

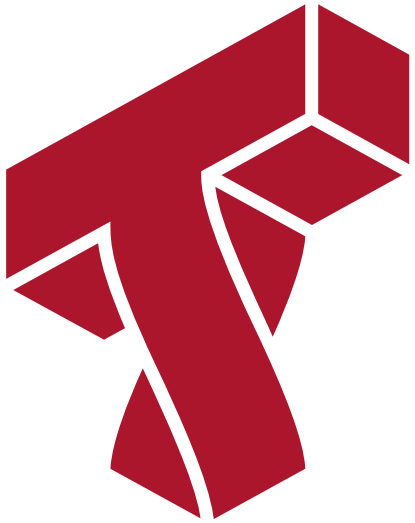
Dataset Size: Should be large enough to estimate future performance: 30% of the data on small tasks, ususally up to not more than a few thousand instances.

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 5: Evaluating Supervised Learning Models

Machine learning algorithms can sometimes fail. How do we assess their performance in a principled way?

When Do We Get Good Performance on Held Out Data?

Suppose you have a classification model trained on images of cats and dogs. On which dataset will it perform better?

- A dataset of German shepherds and Siamese cats?
- A dataset of birds and reptiles?

Clearly the former. Intuitively, ML models are accurate on new data if it is similar to the training data.

Data Distribution

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

$$x, y \sim \mathbb{P} \quad \text{or} \quad \mathcal{D} \sim \mathbb{P}.$$

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

Data Distribution: IID Sampling

The key assumption is that the training examples are *independent and identically 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.

Holdout Dataset

A holdout set

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

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

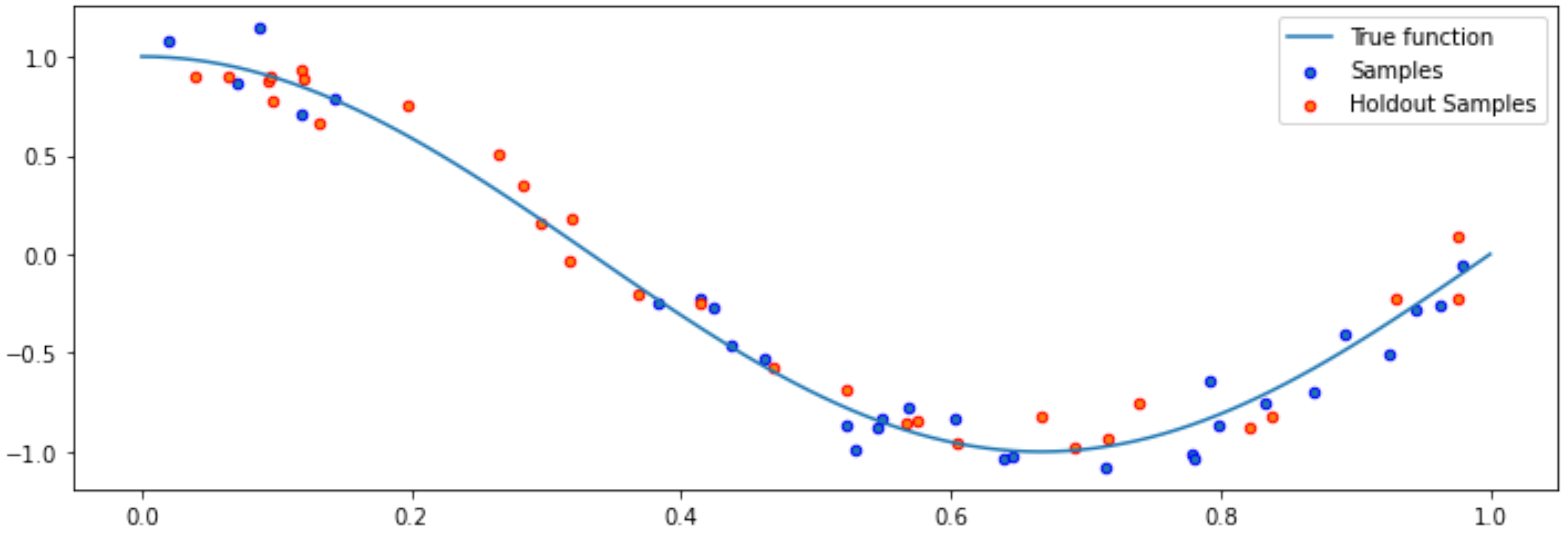
Let's generate a holdout dataset for the example we saw earlier.

```
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()
```

```
<matplotlib.legend.Legend at 0x121440f28>
```



Performance on a Holdout Set

Intuitively, a supervised model f_θ is "good" 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:

- In classification, L is often just accuracy: is $\dot{y}^{(i)} = f_\theta(\dot{x}^{(i)})$?
- L can also implement other metrics: R^2 metric (see Homework 1) for regression, F1 score for document retrieval, etc.

For large enough holdout sets $\dot{\mathcal{D}}$, we will be estimating performance on new data from $\mathbb{P}$.

When Does Supervised Learning Work?

Let's now use these tools to say something about the performance of supervised learning algorithms.

- For large enough training sets $\mathcal{D} \sim \mathbb{P}$, most models will *generalize* to new data from $\mathbb{P}$. We should see the same performance on $\mathcal{D}$ and $\dot{\mathcal{D}} \sim \mathbb{P}$.

- When $\mathcal{D}$ is small, models may overfit $\mathcal{D}$ and perform poorly on new data from $\mathbb{P}$. We can detect this by evaluating them on $\dot{\mathcal{D}}$.