UNIT-II
MACHINE LEARNING

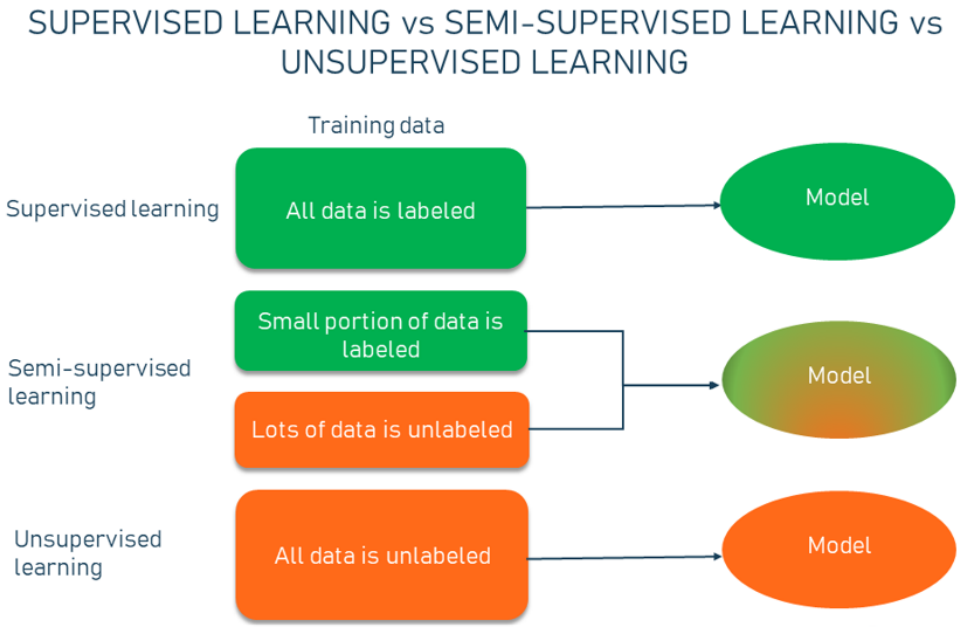
Modeling:-

- The pre-requisite to learn this concept is understand about model, "**A model can be defined as specification of a mathematical relationship that exists between different variables".**
- To give you better understanding let us consider one example, if you are trying to raise money for your social networking site, you might build a business model (like in a spreadsheet) that takes inputs like **"**Number of users", **"**Ad revenue per user"," Number of employees", and it outputs your annual profit for the next several years.
- Another example for model is if you've ever watched poker on television, you know that they estimate each player's "win probability" in real time based on a model that takes into account the cards that have been revealed so far and the distribution of cards in the deck. And the poker model is based on probability theory, the rules of poker, and some reasonably assumptions.

Machine Learning:-

- Machine Learning definition can be varied from person to person, but we will describe it as creating and using models that are learned from data. The real-time applications for machine learning is:

1.) Predicting whether an email is spam or not,
2.) Predicting whether a credit transaction is fraudulent
3.) Predicting which football team is going to win the Super Bowl.



➔ Supervised models (in which there is a set of data labeled with the correct answers to learn from), and unsupervised models (in which there are no such labels). There are various other types like semi supervised (in which only some of the data are labeled).

➔ For instance, we might assume that a person's height is (roughly) a linear function of his weight and then use data to learn what that linear function is. Or we might assume that a decision tree is a good way to diagnose what diseases our patients have and then use data to learn the "optimal" such tree.

OVERFITTING AND UNDERFITTING:-

1.) Overfitting and underfitting are two common problems in machine learning. **'Overfitting occurs when a model is too complex and learns the noise in the training data instead of the underlying pattern."** This leads to poor generalization performance on new data.
2.) "**Underfitting occurs when a model is too simple and cannot capture the underlying pattern of the data".** This also leads to poor generalization performance on new data.

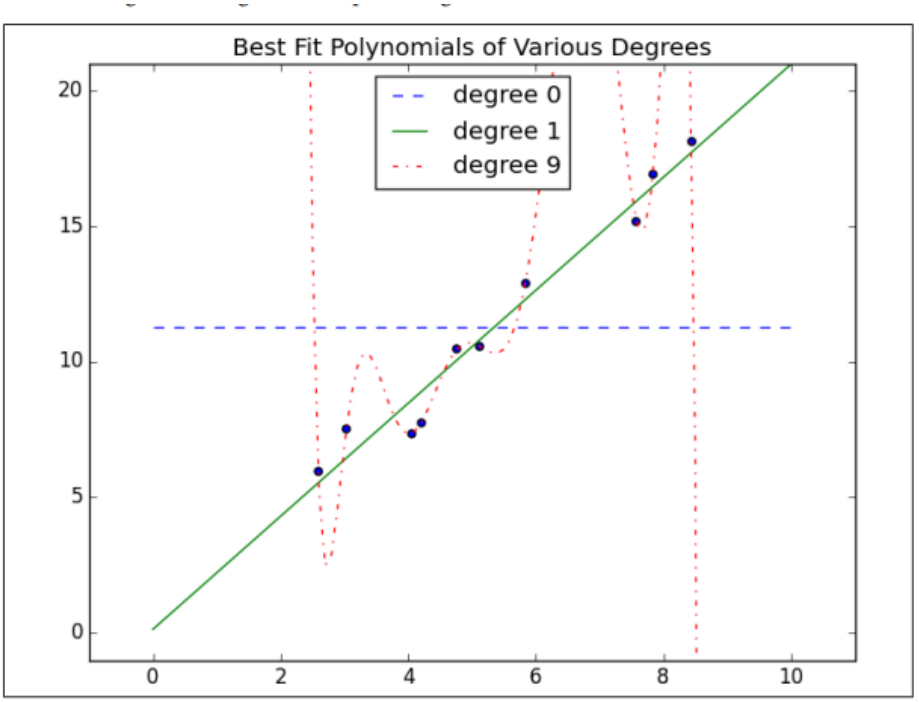


Figure 11-1. Overfitting and underfitting

In the above diagram if you observe a linear regression model is taken which will take height of a person as input and produce their corresponding weight as output.

The horizontal line shows the best fit degree 0 (i.e., constant) polynomial. It severely underfits the Training data.

The best fit degree 9 (i.e., 10-parameter) polynomial goes through every training data point exactly, but it very severely overfits—if we were to pick a few more data points it would quite likely miss them.

The most fundamental approach involves using different data to train the model and to test the model. The simplest way to do this is to split your data set, so that (for example) two-thirds of it is used to train the model, after which we measure the model's performance on the remaining third:

```
def split_data(data, prob):
    """split data into fractions [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results
```

```
def train_test_split(x, y, test_pct):
    data = zip(x, y)                                # pair corresponding values
    train, test = split_data(data, 1 - test_pct)  # split the data set of pairs
    x_train, y_train = zip(*train)                  # magical un-zip trick
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test
```

CORRECTNESS

1.) Imagine building a model to make a binary judgment. Is this email spam? Should we hire this candidate? Is this air traveler secretly a terrorist?
2.) Given a set of labeled data and such a predictive model, every data point lies in one of four categories:

- **True positive:** "This message is spam, and we correctly predicted spam."
- **False positive (Type 1 Error):** "This message is not spam, but we predicted spam."
- **False negative (Type 2 Error):** "This message is spam, but we predicted not spam."
- **True negative:** "This message is not spam, and we correctly predicted not spam."

❖ We often represent these as counts in a confusion matrix:

	Spam	not Spam
predict "Spam"	True Positive	False Positive
predict "Not Spam"	False Negative	True Negative

❖ Metrics to evaluate confusion matrix:-

1.) We will understand this topic by considering an example

	leukemia	no leukemia	total
"Luke"	70	4,930	5,000
not "Luke"	13,930	981,070	995,000
total	14,000	986,000	1,000,000

i.) We can then use these to compute various statistics about model performance. For example, accuracy is defined as the fraction of correct predictions

```
def accuracy(tp, fp, fn, tn):
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

print accuracy(70, 4930, 13930, 981070)     # 0.98114
```

ii.) That seems like a pretty impressive number. But clearly this is not a good test, which means that we probably shouldn't put a lot of credence in raw accuracy.

iii.) It's common to look at the combination of precision and recall. Precision measures how accurate our positive predictions were:

```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

print precision(70, 4930, 13930, 981070)    # 0.014
```

iv.) And recall measures what fraction of the positives our model identified

```
def recall(tp, fp, fn, tn):
    return tp / (tp + fn)

print recall(70, 4930, 13930, 981070)       # 0.005
```

v.) Sometimes precision and recall are combined into the F1 score, which is defined as

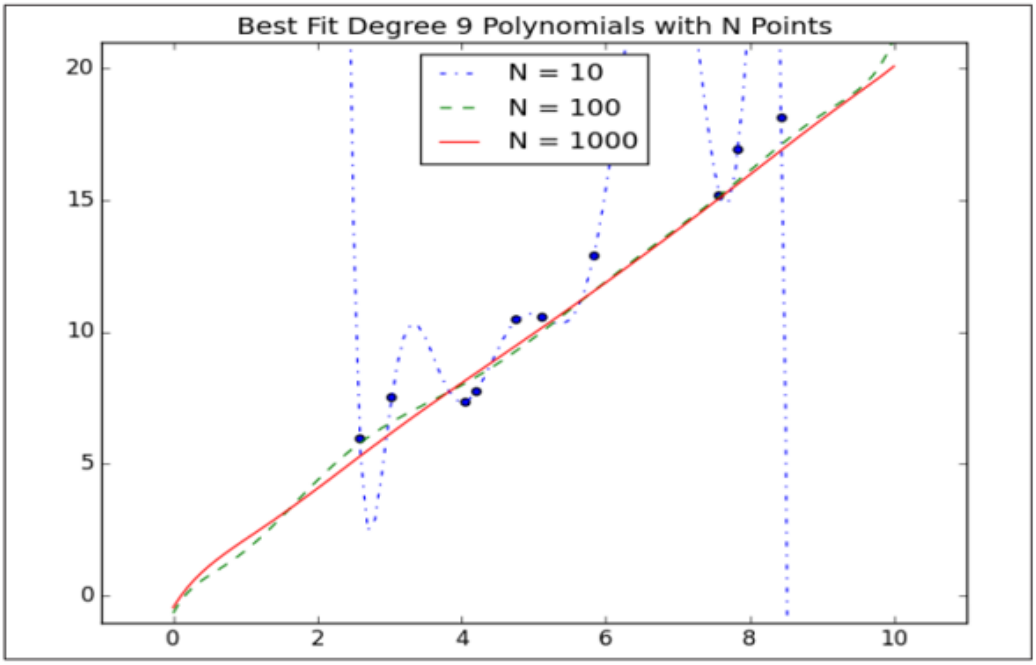
```
def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

❖ This is the harmonic mean of precision and recall and necessarily lies between them

❖ Usually the choice of a model involves a trade-off between precision and recall. A model that predicts "yes" when it's even a little bit confident will probably have a high recall but a low precision; a model that predicts "yes" only when it's extremely confi- dent is likely to have a low recall and a high precision

The Bias-Variance Trade-off

- The bias-variance trade-off is a fundamental concept in data science that relates to the trade-off between the complexity of a model and its ability to generalize to new data.
- A model with high bias is one that makes strong assumptions about the data and tends to underfit it, meaning that it cannot capture the underlying patterns well.
- A model with high variance is one that is very flexible and adapts to the data too much, meaning that it can overfit it, meaning that it cannot generalize well to new data.
- The goal of data science is to find a balance between bias and variance, such that the model can fit the data well and also perform well on new data.
- For example, the degree 0 model in "Overfitting and Underfitting" which is in the earlier concept will make a lot of mistakes for pretty much any training set (drawn from the same population), which means that it has a high bias. However, any two randomly chosen training sets should give pretty similar models (since any two randomly chosen train- ing sets should have pretty similar average values). So we say that it has a low variance. High bias and low variance typically correspond to underfitting
- On the other hand, the degree 9 model fit the training set perfectly. It has very low bias but very high variance (since any two training sets would likely give rise to very different models). This corresponds to overfitting.



Reducing variance with more data

- Going from the degree 0 model in "Overfitting and Underfitting" to the degree 1 model was a big improvement.
- In the above figure, we fit a degree 9 polynomial to different size samples. The model fit based on 10 data points is all over the place, as we saw before. If we instead trained on 100 data points, there's much less overfitting. And the model trained from 1,000 data points looks very similar to the degree 1 model.

Here are some key takeaways from the book about the bias-variance trade-off:

- The bias-variance trade-off is not a property of the model itself, but rather of how well it fits a particular dataset. A model that has low bias and low variance on one dataset might have high bias and high variance on another dataset.
- The bias-variance trade-off depends on the amount and quality of the data available. With more data, we can use more complex models without overfitting. With noisy or irrelevant data, we might need simpler models to avoid fitting the noise.
- The bias-variance trade-off also depends on the choice of hyperparameters, which are parameters that control the complexity or flexibility of the model. For example, in k-nearest neighbours, the number of neighbours k is a hyperparameter that affects the bias and variance of the model. A small k means a more flexible model with low bias and high variance, while a large k means a less flexible model with high bias and low variance.

Feature Extraction and Selection

- Features are whatever inputs we provide to our model
- In the simplest case, features are simply given to you. If you want to predict some- one's salary based on her years of experience, then years of experience is the only
- feature you have

- Things become more interesting as your data becomes more complicated. Imagine trying to build a spam filter to predict whether an email is junk or not. Most models won't know what to do with a raw email, which is just a collection of text. You'll have to extract features.
- For example:
  - Does the email contain the word "Data"?
  - How many times does the letter d appear?
  - What was the domain of the sender?
- The first is simply a yes or no, which we typically encode as a 1 or 0. The second is a number. And the third is a choice from a discrete set of options.
- Pretty much always, we'll extract features from our data that fall into one of these three categories. What's more, the type of features we have constrains the type of models we can use.

➔ The Naive Bayes classifier we'll build is suited to yes-or-no features, like the first one in the preceding list. Regression models, require numeric features (which could include dummy variables that are 0s and 1s). And decision trees, can deal with numeric or categorical data.

➔ For example, your inputs might be vectors of several hundred numbers. Depending on the situation, it might be appropriate to distill these down to handful of important dimensions and use only those small number of features. Or it might be appropriate to use a technique that penalizes models the more features they use.

k-Nearest Neighbors Model:-

Nearest neighbors is one of the simplest predictive models there is. It makes no math- ematical assumptions, and it doesn't require any sort of heavy machinery. The only things it requires are: • Some notion of distance • An assumption that points that are close to one another are similar.

a. Nearest neighbors, on the other hand, quite con- sciously neglects a lot of information, since the prediction for each new point depends only on the handful of points closest to it. In the general situation, we have some data points and we have a corresponding set of labels. The labels could be True and False, indicating whether each input satisfies some condition like "is spam?" or "is poisonous?" or "would be enjoyable to watch?"

b. In our case, the data points will be vectors, which means that we can use the distance function. Let's say we've picked a number k like 3 or 5. Then when we want to classify some new data point, we find the k nearest labeled points.

c. This approach is sure to work eventually, since in the worst case we go all the way down to just one label, at which point that one label wins.

```
def knn_classify(k, labeled_points, new_point):
    """each labeled point should be a pair (point, label)"""

    # order the labeled points from nearest to farthest
    by_distance = sorted(labeled_points,
                         key=lambda (point, _): distance(point, new_point))

    # find the labels for the k closest
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # and let them vote
    return majority_vote(k_nearest_labels)
```

Example:-

KNN can be used for both classification and regression predictive problems. KNN falls in the supervised learning family of algorithms. Informally, this means that we are given a labelled dataset consisting of training observations (x, y) would like to capture the relationship. More formally, our goal is to learn a function h:X→Y

In the classification setting, the K-nearest neighbor algorithm essentially boils down to forming a majority vote between the K most similar instances to a given "unseen" observation. Similarity is defined according to a distance metric between two data points. The k-nearest-neighbor classifier is commonly based on the Euclidean distance between a test sample and the specified training

$$d(x_i, x_l) = \sqrt{(x_{i1} - x_{l1})^2 + (x_{i2} - x_{l2})^2 + \ldots + (x_{ip} - x_{lp})^2}$$

samples.

STEP 1: Choose the number K of neighbors

STEP 2: Take the K nearest neighbors of the new data point, according to your distance metric

STEP 3: Among these K neighbors, count the number of data points to each category

STEP 4: Assign the new data point to the category where you counted the most neighbors

The dataset consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
5	6	5.4	3.9	1.7	0.4	Iris-setosa
6	7	4.6	3.4	1.4	0.3	Iris-setosa
7	8	5.0	3.4	1.5	0.2	Iris-setosa
8	9	4.4	2.9	1.4	0.2	Iris-setosa
9	10	4.9	3.1	1.5	0.1	Iris-setosa

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import cross_val_score
# Importing the dataset
dataset = pd.read_csv('../input/Iris.csv')
dataset.shape
dataset.head(5)
dataset.describe()
dataset.groupby('Species').size()
feature_columns = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm','PetalWidthCm']
X = dataset[feature_columns].values
y = dataset['Species'].values
le = LabelEncoder()
y = le.fit_transform(y)
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

```
# Instantiate learning model (k = 3)
classifier = KNeighborsClassifier(n_neighbors=3)
# Fitting the model
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
cm
```

```
array([[11,  0,  0],
       [ 0, 12,  1],
       [ 0,  0,  6]])
```

```
accuracy = accuracy_score(y_test, y_pred)*100
print('Accuracy of our model is equal ' + str(round(accuracy, 2)) + ' %.')
```

```
Accuracy of our model is equal 96.67 %.
```

NAÏVE BAYES:-

Naive Bayes is a simple and effective classification algorithm that is based on Bayes' theorem and is used for solving classification problems . It is a probabilistic classifier that predicts the probability of an object based on the probability of its features. Naive Bayes is mainly used in text classification, spam filtering, and sentiment analysis. The algorithm is called "naive" because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable.

A Really Dumb Spam Filter

Imagine a "universe" that consists of receiving a message chosen randomly from all possible messages. Let S be the event "the message is spam" and V be the event "the message contains the word "Internship." Then Bayes's Theorem tells us that the probability that the message is spam conditional on containing the word Internship is:

$$P(S \mid V) = [P(V \mid S)P(S)] / [P(V \mid S)P(S) + P(V \mid \neg S)P(\neg S)]$$

The numerator is the probability that a message is spam and contains Internship, while the denominator is just the probability that a message contains Internship. Hence you can think of this calculation as simply representing the proportion of Internship messages that are spam.

If we have a large collection of messages we know are spam, and a large collection of messages we know are not spam, then we can easily estimate below probability.

$$P(V \mid S) \text{ and } P(V \mid \neg S)$$

we further assume that any message is equally likely to be spam or not-spam (so that $P(S) = P(\neg S) = 0.5$), then:

$$P(S|V) = P(V|S)/[P(V|S) + P(V|\neg S)]$$

For example, if 50% of spam messages have the word *viagra*, but only 1% of nonspam messages do, then the probability that any given *viagra*-containing email is spam is:

$$0.5/(0.5 + 0.01) = 98\,\%$$

A More Sophisticated Spam Filter:-

Imagine now that we have a vocabulary of many words w1 , ...,wn . To move this into the realm of probability theory, we'll write Xi for the event "a message contains the word wi Also imagine that we've come up with an estimate P (Xi | S) for the probability that a spam message contains the i^{th} word, and a similar estimate P (Xi | ¬S) for the probability that a non spam message contains the i^{th} word.

The key to Naive Bayes is making the (big) assumption that the presences of each word are independent of one another, conditional on a message being spam or not. Intuitively, this assumption means that knowing whether a certain spam message contains the word "Internship" gives you no information about whether that same message contains the word "rolex." In math terms, this means that:

$$P(X_1 = x_1, \ldots, X_n = x_n|S) = P(X_1 = x_1|S) \times \cdots \times P(X_n = x_n|S)$$

The Naive Bayes assumption allows us to compute each of the probabilities on the right simply by multiplying together the individual probability estimates for each vocabulary word.

In practice, you usually want to avoid multiplying lots of probabilities together, to avoid a problem called underflow, in which computers don't deal well with floating point numbers that are too close to zero. Recalling from algebra that log ab = log a + log b and that exp log x = x, we usually compute p1 pn * ĵ* as the equivalent (but floating-point-friendlier).

$$\exp\left(\log(p_1) + \cdots + \log(p_n)\right)$$

IMPLEMENTATION:-

Now we have all the pieces we need to build our classifier. First, let's create a simple function to tokenize messages into distinct words. We'll first convert each message to lowercase; use re.findall() to extract "words" consisting of letters, numbers, and apostrophes; and finally use set() to get just the distinct words:

```
def tokenize(message):
    message = message.lower()                       # convert to lowercase
    all_words = re.findall("[a-z0-9']+", message)   # extract the words
    return set(all_words)                           # remove duplicates
```

Our second function will count the words in a labeled training set of messages. We'll have it return a dictionary whose keys are words, and whose values are two-element lists [spam_count, non_spam_count] corresponding to how many times we saw that word in both spam and nonspam messages.

```
def count_words(training_set):
    """training set consists of pairs (message, is_spam)"""
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in tokenize(message):
            counts[word][0 if is_spam else 1] += 1
    return counts
```

Our next step is to turn these counts into estimated probabilities using the smoothing we described before. Our function will return a list of triplets containing each word, the probability of seeing that word in a spam message, and the probability of seeing that word in a non-spam message.

```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
    """turn the word_counts into a list of triplets
    w, p(w | spam) and p(w | ~spam)"""
    return [(w,
             (spam + k) / (total_spams + 2 * k),
             (non_spam + k) / (total_non_spams + 2 * k))
             for w, (spam, non_spam) in counts.iteritems()]
```

The last piece is to use these word probabilities (and our Naive Bayes assumptions) to assign probabilities to messages.

Gradient Descent:-

Suppose we have some function f that takes as input a vector of real numbers and outputs a single real number. One simple such function is:

```
def sum_of_squares(v):
    """computes the sum of squared elements in v"""
    return sum(v_i ** 2 for v_i in v)
```

We'll frequently need to maximize (or minimize) such functions. That is, we need to find the input v that produces the largest (or smallest) possible value. For functions like ours, the gradient (if you remember your calculus, this is the vector of partial derivatives) gives the

input direction in which the function most quickly increases. (If you don't remember your calculus, take my word for it or look it up on the Internet.

Accordingly, one approach to maximizing a function is to pick a random starting point, compute the gradient, take a small step in the direction of the gradient (i.e., the direction that causes the function to increase the most), and repeat with the new starting point. Similarly, you can try to minimize a function by taking small steps in the opposite direction

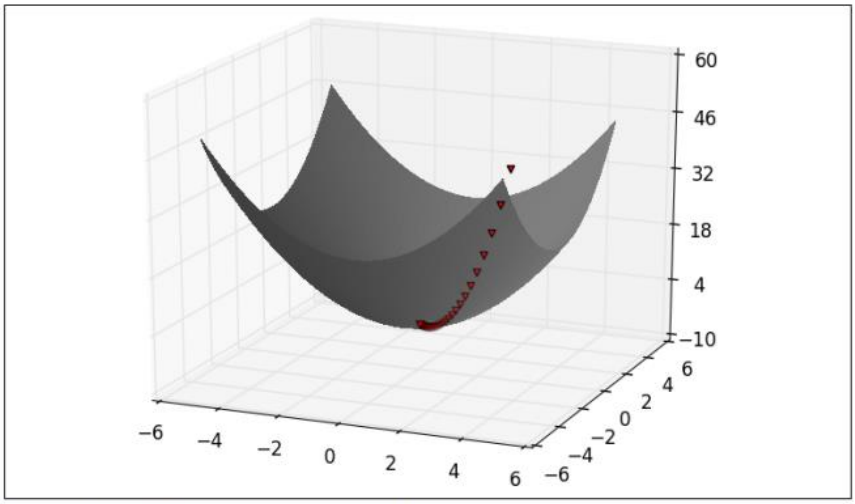


Figure 8-1. Finding a minimum using gradient descent

Estimating the Gradient

If f is a function of one variable, its derivative at a point x measures how f(x) changes when we make a very small change to x. It is defined as the limit of the difference quotients:

```
def difference_quotient(f, x, h):
    return (f(x + h) - f(x)) / h
```

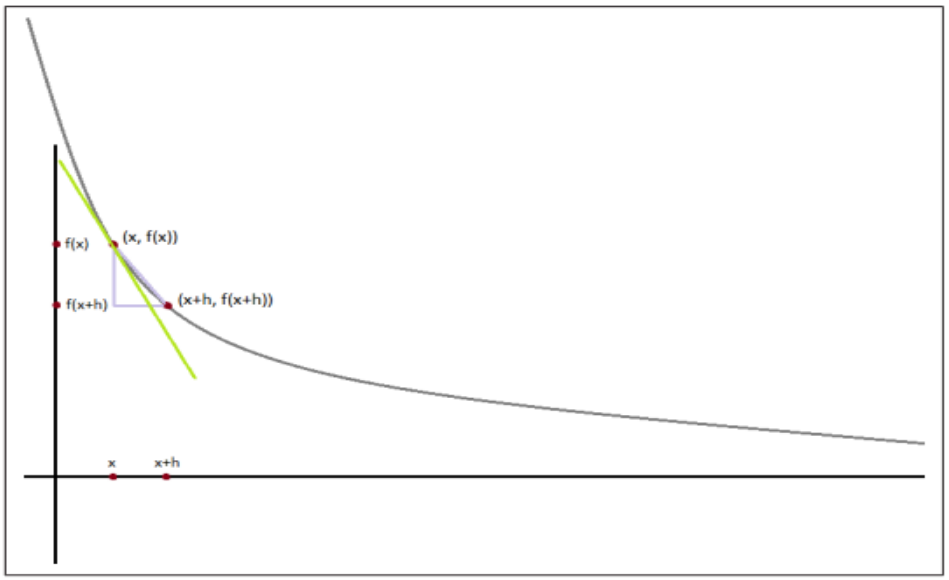


Figure 8-2. Approximating a derivative with a difference quotient

The derivative is the slope of the tangent line at x, f x , while the difference quotient is the slope of the not-quite-tangent line that runs through x + h, f x + h . As h gets smaller and smaller, the not-quite-tangent line gets closer and closer to the tan- gent line

For many functions it's easy to exactly calculate derivatives. For example, the `square` function:

```
def square(x):
    return x * x
```

has the derivative:

```
def derivative(x):
    return 2 * x
```

A major drawback to this "estimate using difference quotients" approach is that it's computationally expensive. If v has length n, estimate gradient has to evaluate f on 2n different inputs. If you're repeatedly estimating gradients, you're doing a whole lot of extra work.

Stochastic Gradient Descent

we'll be using gradient descent to choose the parameters of a model in a way that minimizes some notion of error. Using the previous batch approach, each gradient step requires us to make a prediction and compute the gradient for the whole data set, which makes each step take a long time. Now, usually these error functions are additive, which means that the predictive error on the whole data set is simply the sum of the predictive errors for each data point. The stochastic version will typically be a lot faster than the batch version. Of course, we'll want a version that maximizes as well.