Lecture 7: Generative Algorithms

Applied Machine Learning

Volodymyr Kuleshov
Cornell Tech

Part 1: Generative Models

In this lecture, we are going to look at generative algorithms and their applications to classification.

We will start by defining the concept of a generative *model*.

Review: Components of A Supervised Machine Learning Problem

At a high level, a supervised machine learning problem has the following structure:

$$\underbrace{\text{Training Dataset}}_{\text{Attributes + Features}} + \underbrace{\text{Learning Algorithm}}_{\text{Model Class + Objective + Optimizer}} \to \text{Predictive Model}$$

Review: Probabilistic Models

A (parametric) probabilistic model with parameters θ is a probability distribution

$$P_\theta(x, y) : \mathcal{X} \times \mathcal{Y} \to [0, 1].$$

This model can approximate the data distribution $\mathbb{P}(x, y)$.

If we know $P_\theta(x, y)$, we can compute predictions using the formula

$$P_\theta(y|x) = \frac{P_\theta(x, y)}{P_\theta(x)} = \frac{P_\theta(x, y)}{\sum_{y \in \mathcal{Y}} P_\theta(x, y)}.$$

Review: Maximum Likelihood Learning

In order to fit probabilistic models, we use the following objective:

$$\max_{\theta} \mathbb{E}_{x,y\sim\mathbb{P}_{\text{data}}} \log P_{\theta}(x, y).$$

This seeks to find a model that assigns high probability to the training data.

Review: Conditional Probabilistic Models

Alternatively, we may define a model of the conditional probability distribution:

$$P_\theta(y|x) : \mathcal{X} \times \mathcal{Y} \to [0, 1].$$

These are trained using conditional maximum likelihood:

$$\max_\theta \mathbb{E}_{x,y\sim\mathbb{P}_{\text{data}}} \log P_\theta(y|x).$$

This seeks to find a model that assigns high conditional probability to the target y for each x.

Logistic regression is an example of this approach.

Discriminative vs. Generative Models

These two types of models are also known as *generative* and *discriminative*.

$$\underbrace{P_\theta(x, y) : \mathcal{X} \times \mathcal{Y} \to [0, 1]}_{\text{generative model}} \qquad \underbrace{P_\theta(y|x) : \mathcal{X} \times \mathcal{Y} \to [0, 1]}_{\text{discriminative model}}$$

- The models parametrize different kinds of probabilities
- They involve different training objectives and make different predictions
- Their uses are different (e.g., prediction, generation); more later!

Classification Dataset: Iris Flowers

To demonstrate the two approaches, we are going to use the Iris flower dataset.

It's a classical dataset originally published by [R. A. Fisher (https://en.wikipedia.org/wiki/Ronald_Fisher)](https://en.wikipedia.org/wiki/Ronald_Fisher) in 1936. Nowadays, it's widely used for demonstrating machine learning algorithms.

In [1]:
```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
from sklearn import datasets

# Load the Iris dataset
iris = datasets.load_iris(as_frame=True)

# print part of the dataset
iris_X, iris_y = iris.data, iris.target
pd.concat([iris_X, iris_y], axis=1).head()
```

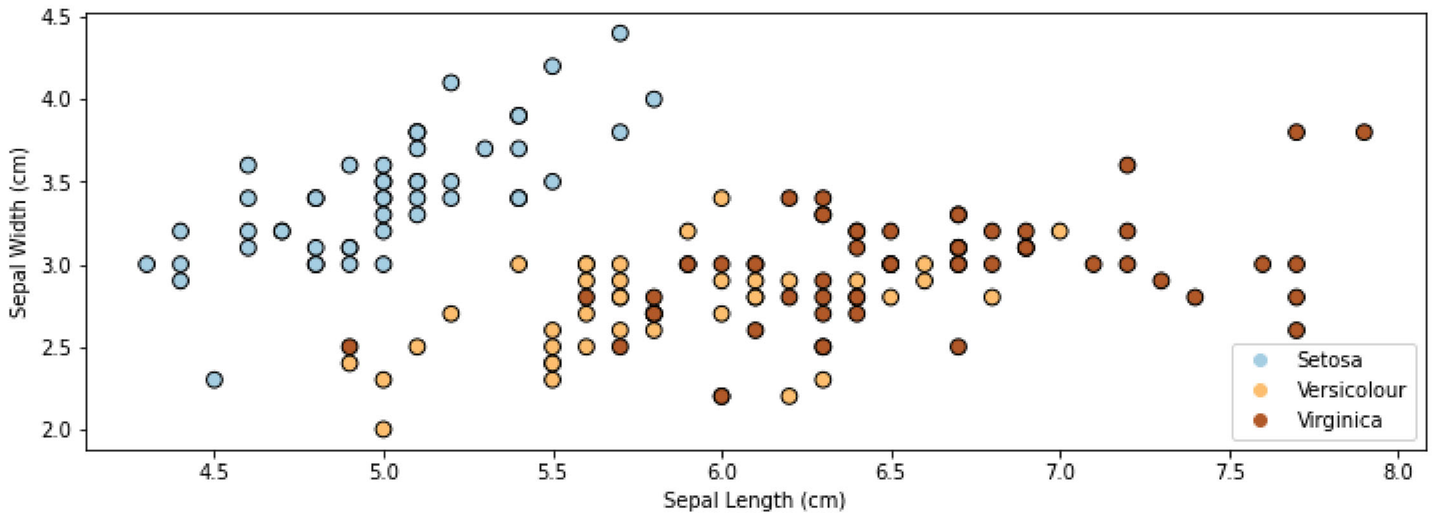
Out[1]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

If we only consider the first two feature columns, we can visualize the dataset in 2D.

In [2]:
```python
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html
%matplotlib inline
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

# create 2d version of dataset
X = iris_X.to_numpy()[:,:2]
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

# Plot also the training points
p1 = plt.scatter(X[:, 0], X[:, 1], c=iris_y, edgecolor='k', s=60, cmap=plt.cm.Pair
ed)
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.legend(handles=p1.legend_elements()[0], labels=['Setosa', 'Versicolour', 'Virg
inica'], loc='lower right')
```

Out[2]:
```
<matplotlib.legend.Legend at 0x124f39cc0>
```



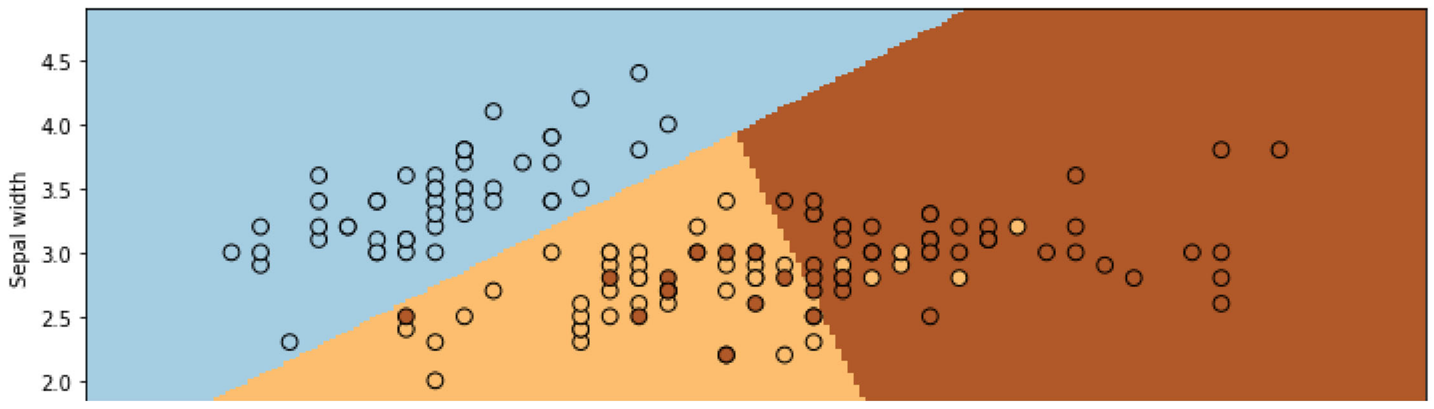
Example: Discriminative Model

An example of a discriminative model is logistic or softmax regression.

- Discriminative models directly partition the feature space into regions associated with each class and separated by a decision boundary.
- Given features x, discriminative models directly map to predicted classes (e.g., via the function $\sigma(\theta^\top x)$ for logistic regression).

In [3]:
```
# https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.ht
ml
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(C=1e5, multi_class='multinomial')

# Create an instance of Softmax and fit the data.
logreg.fit(X, iris_y)
xx, yy = np.meshgrid(np.arange(x_min, x_max, .02), np.arange(y_min, y_max, .02))
Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=iris_y, edgecolors='k', s=60, cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
```

Out[3]:
```
Text(0, 0.5, 'Sepal width')
```



Example: Generative Model

Generative modeling can be seen as taking a different approach:

1. In the Iris example, we first build a model of how each type of flower looks, i.e. we can learn the distribution
$$p(x|y=k) \text{ for each class } k.$$
It defines a model of how each flower is *generated*, hence the name.

1. Given a new flower datapoint x', we can match it against each flower model and find the type of flower that looks most similar to it. Mathematically, this corresponds to:

$$\begin{aligned}\arg\max_y \log p(y|x) &= \arg\max_y \log \frac{p(x|y)p(y)}{p(x)} \\ &= \arg\max_y \log p(x|y)p(y),\end{aligned}$$

   where we have applied Bayes' rule in the first line.

Generative vs. Discriminative Approaches

How do we know which approach is better?

- If we only care about prediction, we don't need a model of $P(x)$. We can solve precisely the problem we care about.
    - Discriminative models will often be more accurate.

- If we care about other tasks (generation, dealing with missing values, etc.) or if we know the true model is generative, we want to use the generative approach.

More on this later!

Part 2: Gaussian Discriminant Analysis

We are now going to continue our discussion of classification.

- We will see a new classification algorithm, Gaussian Discriminant Analysis.
- This will be our first example of generative machine learning model.

Review: Classification

Consider a training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

We distinguish between two types of supervised learning problems depnding on the targets $y^{(i)}$.

1. **Regression**: The target variable $y \in \mathcal{Y}$ is continuous: $\mathcal{Y} \subseteq \mathbb{R}$.
2. **Classification**: The target variable y is discrete and takes on one of K possible values: $\mathcal{Y} = \{y_1, y_2, \ldots y_K\}$. Each discrete value corresponds to a *class* that we want to predict.

Review: Generative Models

There are two types of probabilistic models: *generative* and *discriminative*.

$$\underbrace{P_\theta(x, y) : \mathcal{X} \times \mathcal{Y} \to [0, 1]}_{\text{generative model}} \qquad \underbrace{P_\theta(y|x) : \mathcal{X} \times \mathcal{Y} \to [0, 1]}_{\text{discriminative model}}$$

- They involve different training objectives and make different predictions
- Their uses are different (e.g., prediction, generation); more later!

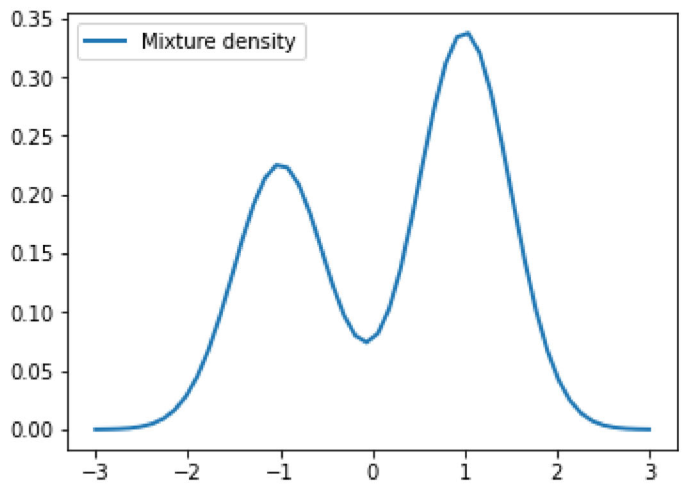
Mixtures of Gaussians

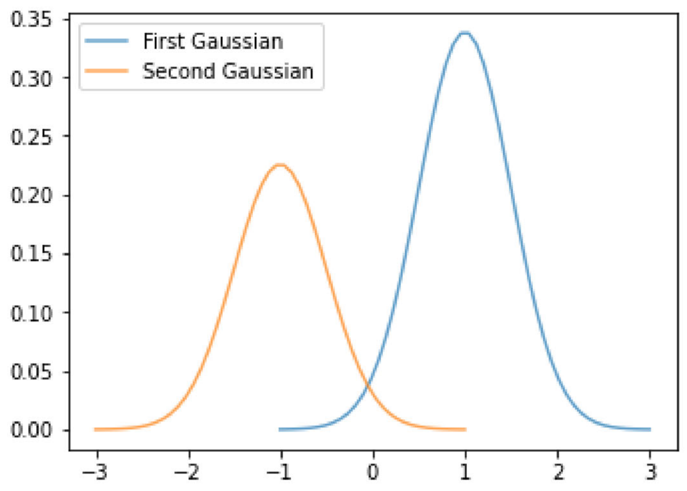
A mixture of K Gaussians is a distribution $P(x)$ of the form:

$$\phi_1 \mathcal{N}(x; \mu_1, \Sigma_1) + \phi_2 \mathcal{N}(x; \mu_2, \Sigma_2) + \ldots + \phi_K \mathcal{N}(x; \mu_K, \Sigma_K).$$

- Each $\mathcal{N}(x; \mu_k, \Sigma_k)$ is a (multivariate) Gaussian distribution with mean μ_k and covariance Σ_k.
- The ϕ_k are weights, and the above sum is a weighted average of the K Gaussians.

We can easily visualize this in 1D:

In [4]:
```
def N(x,mu,sigma):
    return np.exp(-.5*(x-mu)**2/sigma**2)/np.sqrt(2*np.pi*sigma)

def mixture(x):
    return 0.6*N(x,mu=1,sigma=0.5) + 0.4*N(x,mu=-1,sigma=0.5)

xs, xs1, xs2 = np.linspace(-3,3), np.linspace(-1,3), np.linspace(-3,1)
plt.subplot('121')
plt.plot(xs, mixture(xs), label='Mixture density', linewidth=2)
plt.legend()

plt.subplot('122')
plt.plot(xs1, 0.6*N(xs1,mu=1,sigma=0.5), label='First Gaussian', alpha=0.7)
plt.plot(xs2, 0.4*N(xs2,mu=-1,sigma=0.5), label='Second Gaussian', alpha=0.7)
plt.legend()
```
Out[4]:
```
<matplotlib.legend.Legend at 0x125bd5470>
```




Gaussian Discriminant Model

We may use this approach to define a model P_θ. This will be the basis of an algorthim called Gaussian Discriminant Analysis.

- The distribution over classes is [Categorical (https://en.wikipedia.org/wiki/Categorical_distribution)](https://en.wikipedia.org/wiki/Categorical_distribution), denoted $\text{Categorical}(\phi_1, \phi_2, \ldots, \phi_K)$. Thus, $P_\theta(y = k) = \phi_k$.
- The conditional probability $P_\theta(x \mid y = k)$ of the data under class k is a [multivariate Gaussian (https://en.wikipedia.org/wiki/Multivariate_normal_distribution)](https://en.wikipedia.org/wiki/Multivariate_normal_distribution) $\mathcal{N}(x; \mu_k, \Sigma_k)$ with mean and covariance μ_k, Σ_k.

Thus, $P_\theta(x, y)$ is a mixture of K Gaussians:

$$P_\theta(x, y) = \sum_{k=1}^K P_\theta(y = k) P_\theta(x|y = k) = \sum_{k=1}^K \phi_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

Intuitively, this model defines a story for how the data was generated. To obtain a data point,

- First, we sample a class $y \sim \text{Categorical}(\phi_1, \phi_2, \ldots, \phi_K)$ with class proportions given by the ϕ_k.
- Then, we sample an x from a Gaussian distribution $\mathcal{N}(\mu_k, \Sigma_k)$ specific to that class.

Such a story can be constructed for most generative algorithms and helps understand them.

Classification Dataset: Iris Flowers

To demonstrate this approach, we are going to use the Iris flower dataset.

It's a classical dataset originally published by R. A. Fisher (https://en.wikipedia.org/wiki/Ronald_Fisher) in 1936. Nowadays, it's widely used for demonstrating machine learning algorithms.

In [5]:
```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
from sklearn import datasets

# Load the Iris dataset
iris = datasets.load_iris(as_frame=True)

# print part of the dataset
iris_X, iris_y = iris.data, iris.target
pd.concat([iris_X, iris_y], axis=1).head()
```

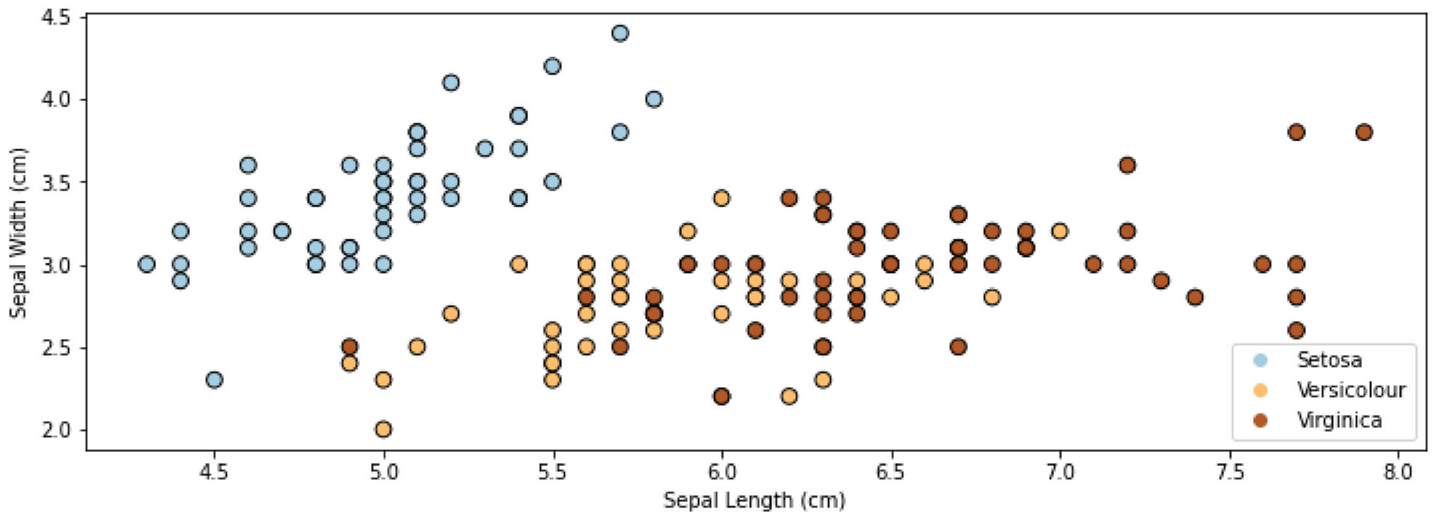
Out[5]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

If we only consider the first two feature columns, we can visualize the dataset in 2D.

```
In [6]: # https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html
        %matplotlib inline
        from matplotlib import pyplot as plt
        plt.rcParams['figure.figsize'] = [12, 4]

        # create 2d version of dataset
        X = iris_X.to_numpy()[:,:2]
        x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
        y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

        # Plot also the training points
        p1 = plt.scatter(X[:, 0], X[:, 1], c=iris_y, edgecolor='k', s=60, cmap=plt.cm.Pair
        ed)
        plt.xlabel('Sepal Length (cm)')
        plt.ylabel('Sepal Width (cm)')
        plt.legend(handles=p1.legend_elements()[0], labels=['Setosa', 'Versicolour', 'Virg
        inica'], loc='lower right')
```

```
Out[6]: <matplotlib.legend.Legend at 0x125c4af28>
```



Example: Iris Flower Classification

Let's see how this approach can be used in practice on the Iris dataset.

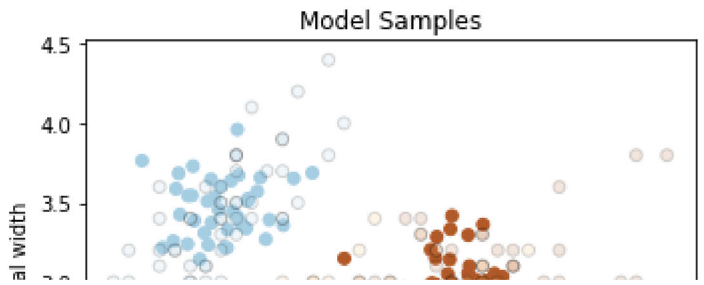
- We will "guess" a good set of parameters for a Gaussian Discriminant model
- We will sample from the model and compare to the true data

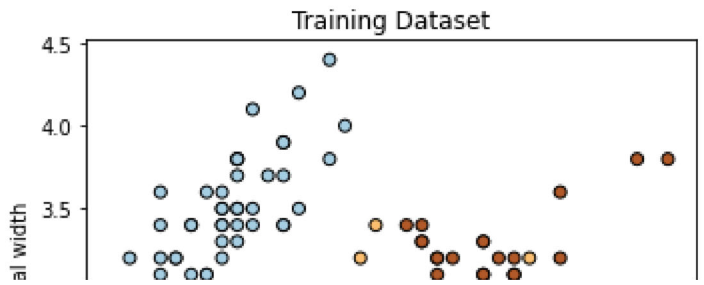
In [7]:
```
s = 100 # number of samples
K = 3 # number of classes
d = 2 # number of features

# guess the parameters
phi = 1./K * np.ones(K,)
mus = np.array(
[[5.0, 3.5],
 [6.0, 2.5],
 [6.5, 3.0]]
)
Sigmas = 0.05*np.tile(np.reshape(np.eye(2),(1,2,2)),(K,1,1))

# generate data from this model
ys = np.random.multinomial(n=1, pvals=phi, size=(s,)).argmax(axis=1)
xs = np.zeros([s,d])
for k in range(K):
    nk = (ys==k).sum()
    xs[ys==k,:] = np.random.multivariate_normal(mus[k], Sigmas[k], size=(nk,))

print(xs[:10])
```

```
[[6.05480188 2.57822945]
 [5.31460491 3.3924932 ]
 [6.06002739 2.49449373]
 [6.70405162 3.36279592]
 [5.87442218 2.6286033 ]
 [6.61493341 3.0305957 ]
 [4.70751809 3.58818661]
 [5.10663152 3.95995748]
 [4.78309822 3.23922458]
 [5.59456967 3.68846231]]
```

In [8]:
```
plt.subplot('121')
plt.title('Model Samples')
plt.scatter(xs[:,0], xs[:,1], c=ys, cmap=plt.cm.Paired)
plt.scatter(X[:, 0], X[:, 1], c=iris_y, edgecolors='k', cmap=plt.cm.Paired, alpha=
0.15)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

# Plot also the training points
plt.subplot('122')
plt.title('Training Dataset')
plt.scatter(X[:, 0], X[:, 1], c=iris_y, edgecolors='k', cmap=plt.cm.Paired, alpha=
1)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
```

Out[8]:
```
Text(0, 0.5, 'Sepal width')
```





- Our Gaussian Discirminant model generates data that looks not unlike the real data.
- Let's now see how we can learn parameters from data and use the model to make predictions.

Part 3: Gaussian Discriminant Analysis: Learning

We continue our discussion of Gaussian Discriminant analysis, and look at:

- How to learn parameters of the mixture model
- How to use the model to make predictions

Review: Classification

Consider a training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

We distinguish between two types of supervised learning problems depnding on the targets $y^{(i)}$.

1. **Regression**: The target variable $y \in \mathcal{Y}$ is continuous: $\mathcal{Y} \subseteq \mathbb{R}$.
2. **Classification**: The target variable y is discrete and takes on one of K possible values: $\mathcal{Y} = \{y_1, y_2, \ldots y_K\}$. Each discrete value corresponds to a *class* that we want to predict.

Review: Gaussian Discriminant Model

We may define a model P_θ as follows. This will be the basis of an algorthim called Gaussian Discriminant Analysis.

- The distribution over classes is Categorical (https://en.wikipedia.org/wiki/Categorical_distribution), denoted $\text{Categorical}(\phi_1, \phi_2, \ldots, \phi_K)$. Thus, $P_\theta(y = k) = \phi_k$.
- The conditional probability $P(x \mid y = k)$ of the data under class k is a multivariate Gaussian (https://en.wikipedia.org/wiki/Multivariate_normal_distribution) $\mathcal{N}(x; \mu_k, \Sigma_k)$ with mean and covariance μ_k, Σ_k.

Thus, $P_\theta(x, y)$ is a mixture of K Gaussians:

$$P_\theta(x, y) = \sum_{k=1}^K P_\theta(y = k) P_\theta(x | y = k) = \sum_{k=1}^K \phi_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

Review: Maximum Likelihood Learning

In order to fit probabilistic models, we use the following objective:

$$\max_\theta \mathbb{E}_{x,y \sim \mathbb{P}_{\text{data}}} \log P_\theta(x, y).$$

This seeks to find a model that assigns high probability to the training data.

Let's use maximum likelihood to fit the Guassian Discriminant model. Note that model parameterss θ are the union of the parameters of each sub-model:

$$\theta = (\mu_1, \Sigma_1, \phi_1, \ldots, \mu_K, \Sigma_K, \phi_K).$$

Optimizing the Log Likelihood

Given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, \ldots, n\}$, we want to optimize the log-likelihood $\ell(\theta)$:

$$\begin{aligned}\ell(\theta) &= \sum_{i=1}^{n} \log P_\theta(x^{(i)}, y^{(i)}) = \sum_{i=1}^{n} \log P_\theta(x^{(i)}|y^{(i)}) + \sum_{i=1}^{n} \log P_\theta(y^{(i)}) \\ &= \sum_{k=1}^{K} \underbrace{\sum_{i:y^{(i)}=k} \log P(x^{(i)}|y^{(i)}; \mu_k, \Sigma_k)}_{\text{all the terms that involve } \mu_k, \Sigma_k} + \underbrace{\sum_{i=1}^{n} \log P(y^{(i)}; \vec{\phi})}_{\text{all the terms that involve } \vec{\phi}} \quad .\end{aligned}$$

Notice that each set of parameters (μ_k, Σ_k) is found in only one term of the summation over the K classes and the ϕ_k are also in the same term.

Since each (μ_k, Σ_k) for $k = 1, 2, \ldots, K$ is found in one term, optimization over (μ_k, Σ_k) can be carried out independently of all the other parameters by just looking at that term:

$$\begin{aligned}\max_{\mu_k, \Sigma_k} \sum_{i=1}^{n} \log P_\theta(x^{(i)}, y^{(i)}) &= \max_{\mu_k, \Sigma_k} \sum_{l=1}^{K} \sum_{i:y^{(i)}=l} \log P_\theta(x^{(i)} | y^{(i)}; \mu_l, \Sigma_l) \\ &= \max_{\mu_k, \Sigma_k} \sum_{i:y^{(i)}=k} \log P_\theta(x^{(i)} | y^{(i)}; \mu_k, \Sigma_k).\end{aligned}$$

Similarly, optimizing for $\vec{\phi} = (\phi_1, \phi_2, \ldots, \phi_K)$ only involves a single term:

$$\max_{\vec{\phi}} \sum_{i=1}^{n} \log P_\theta(x^{(i)}, y^{(i)}; \theta) = \max_{\vec{\phi}} \sum_{i=1}^{n} \log P_\theta(y^{(i)}; \vec{\phi}).$$

Optimizing the Class Probabilities

These observations greatly simplify the optimization of the model. Let's first consider the optimization over $\vec{\phi} = (\phi_1, \phi_2, \ldots, \phi_K)$. From the previous anaylsis, our objective $J(\vec{\phi})$ equals

$$\begin{aligned} J(\vec{\phi}) &= \sum_{i=1}^{n} \log P_\theta(y^{(i)}; \vec{\phi}) \\ &= \sum_{i=1}^{n} \log \phi_{y^{(i)}} - n \cdot \log \sum_{k=1}^{K} \phi_k \\ &= \sum_{k=1}^{K} \sum_{i:y^{(i)}=k} \log \phi_k - n \cdot \log \sum_{k=1}^{K} \phi_k \end{aligned}$$

Taking the derivative and setting it to zero, we obtain

$$\frac{\phi_k}{\sum_l \phi_l} = \frac{n_k}{n}$$

for each k, where $n_k = |\{i : y^{(i)} = k\}|$ is the number of training targets with class k.

Thus, the optimal ϕ_k is just the proportion of data points with class k in the training set!

Optimizing Conditional Probabilities

Similarly, we can maximize the likelihood

$$\max_{\mu_k, \Sigma_k} \sum_{i:y^{(i)}=k} \log P(x^{(i)} | y^{(i)}; \mu_k, \Sigma_k) = \max_{\mu_k, \Sigma_k} \sum_{i:y^{(i)}=k} \log \mathcal{N}(x^{(i)} | \mu_k, \Sigma_k)$$

over the Gaussian parameters.

Computing the derivative and setting it to zero, we obtain closed form solutions:

$$\mu_k = \frac{\sum_{i:y^{(i)}=k} x^{(i)}}{n_k}$$

$$\Sigma_k = \frac{\sum_{i:y^{(i)}=k} (x^{(i)} - \mu_k)(x^{(i)} - \mu_k)^\top}{n_k}$$

These are just the empirical means and covariances of each class.

Querying the Model

How do we ask the model for predictions? As discussed earler, we can apply Bayes' rule:

$$\arg\max_y P_\theta(y|x) = \arg\max_y P_\theta(x|y)P(y).$$

Thus, we can estimate the probability of x and under each $P_\theta(x|y=k)P(y=k)$ and choose the class that explains the data best.

Classification Dataset: Iris Flowers

To demonstrate this approach, we are going to use the Iris flower dataset.

It's a classical dataset originally published by R. A. Fisher (https://en.wikipedia.org/wiki/Ronald_Fisher) in 1936. Nowadays, it's widely used for demonstrating machine learning algorithms.

In [9]:
```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
from sklearn import datasets

# Load the Iris dataset
iris = datasets.load_iris(as_frame=True)

# print part of the dataset
iris_X, iris_y = iris.data, iris.target
pd.concat([iris_X, iris_y], axis=1).head()
```

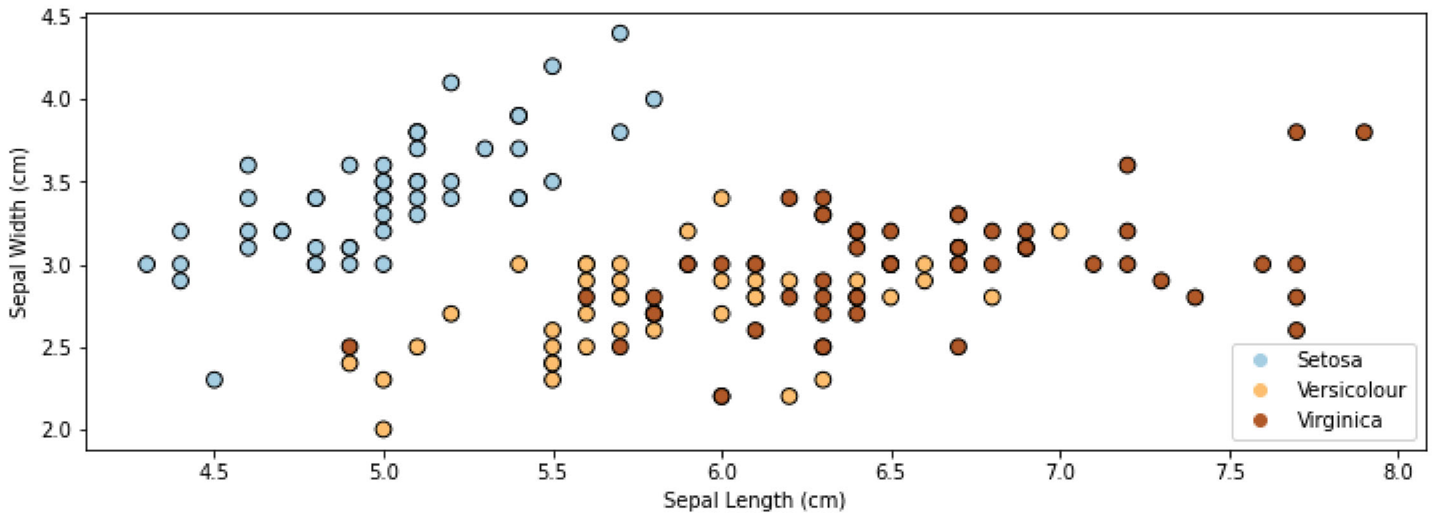
Out[9]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

If we only consider the first two feature columns, we can visualize the dataset in 2D.

In [10]:
```
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html
%matplotlib inline
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

# create 2d version of dataset
X = iris_X.to_numpy()[:,:2]
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

# Plot also the training points
p1 = plt.scatter(X[:, 0], X[:, 1], c=iris_y, edgecolor='k', s=60, cmap=plt.cm.Pair
ed)
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.legend(handles=p1.legend_elements()[0], labels=['Setosa', 'Versicolour', 'Virg
inica'], loc='lower right')
```

Out[10]:
```
<matplotlib.legend.Legend at 0x124dfd278>
```



Example: Iris Flower Classification

Let's see how this approach can be used in practice on the Iris dataset.

- We will learn a good set of parameters for a Gaussian Discriminant model
- We will compare the outputs to the true predictions.

Let's first start by computing the true parameters on our dataset.

In [11]:
```
# we can implement these formulas over the Iris dataset
d = 2 # number of features in our toy dataset
K = 3 # number of clases
n = X.shape[0] # size of the dataset

# these are the shapes of the parameters
mus = np.zeros([K,d])
Sigmas = np.zeros([K,d,d])
phis = np.zeros([K])

# we now compute the parameters
for k in range(3):
    X_k = X[iris_y == k]
    mus[k] = np.mean(X_k, axis=0)
    Sigmas[k] = np.cov(X_k.T)
    phis[k] = X_k.shape[0] / float(n)

# print out the means
print(mus)
```

```
[[5.006 3.428]
 [5.936 2.77 ]
 [6.588 2.974]]
```

We can compute predictions using Bayes' rule.

In [16]:
```
# we can implement this in numpy
def gda_predictions(x, mus, Sigmas, phis):
    """This returns class assignments and p(y|x) under the GDA model.

    We compute \arg\max_y p(y|x) as \arg\max_y p(x|y)p(y)
    """
    # adjust shapes
    n, d = x.shape
    x = np.reshape(x, (1, n, d, 1))
    mus = np.reshape(mus, (K, 1, d, 1))
    Sigmas = np.reshape(Sigmas, (K, 1, d, d))

    # compute probabilities
    py = np.tile(phis.reshape((K,1)), (1,n)).reshape([K,n,1,1])
    pxy = (
        np.sqrt(np.abs((2*np.pi)**d*np.linalg.det(Sigmas))).reshape((K,1,1,1))
        * -.5*np.exp(
            np.matmul(np.matmul((x-mus).transpose([0,1,3,2]), np.linalg.inv(Sigmas
)), x-mus)
        )
    )
    pyx = pxy * py
    return pyx.argmax(axis=0).flatten(), pyx.reshape([K,n])

idx, pyx = gda_predictions(X, mus, Sigmas, phis)
print(idx)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0 0 0 0 0 2 2 2 1 2 1 2 1 2 1 1 1 1 1 1 2 1 1 1 1 1 1 2 1
 2 2 2 2 1 1 1 1 1 1 1 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 2 2 1 2 2 2 2
 2 2 1 1 2 2 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 1 1 2 2 2 1 2 2 2 1 2 2 2 2 2
 2 1]
```

We visualize the decision boundaries like we did earlier.

```
In [14]: from matplotlib.colors import LogNorm
         xx, yy = np.meshgrid(np.arange(x_min, x_max, .02), np.arange(y_min, y_max, .02))
         Z, pyx = gda_predictions(np.c_[xx.ravel(), yy.ravel()], mus, Sigmas, phis)
         logpy = np.log(-1./3*pyx)

         # Put the result into a color plot
         Z = Z.reshape(xx.shape)
         contours = np.zeros([K, xx.shape[0], xx.shape[1]])
         for k in range(K):
             contours[k] = logpy[k].reshape(xx.shape)
         plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)
         for k in range(K):
             plt.contour(xx, yy, contours[k], levels=np.logspace(0, 1, 1))

         # Plot also the training points
         plt.scatter(X[:, 0], X[:, 1], c=iris_y, edgecolors='k', cmap=plt.cm.Paired)
         plt.xlabel('Sepal length')
         plt.ylabel('Sepal width')

         plt.show()
```

Algorithm: Gaussian Discriminant Analysis

- **Type**: Supervised learning (multi-class classification)
- **Model family**: Mixtures of Gaussians.
- **Objective function**: Log-likelihood.
- **Optimizer**: Closed form solution.

Special Cases of GDA

Many important generative algorithms are special cases of Gaussian Discriminative Analysis

- Linear discriminant analysis (LDA): all the covariance matrices Σ_k take the same value.
- Gaussian Naive Bayes: all the covariance matrices Σ_k are diagonal.
- Quadratic discriminant analysis (QDA): another term for GDA.

Generative vs. Discriminative Approaches

Pros of discriminative models:

- Often more accurate because they make fewer modeling assumptions.

Pros of generative models:

- Can do more than just prediction: generation, fill-in missing features, etc.
- Can include extra prior knowledge; if prior knowledge is correct, model will be more accurate.
- Often have closed-form solutions, hence are faster to train.

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

In [10]:
```
# slow:
out = np.zeros([2000,2000])
for i in range(2000):
    for j in range(2000):
        out[i,j] = np.linalg.norm(X[i] - Y[j])

# fast
# ??
```

In [12]:
```
# fast:
out = X.dot(theta)

# slow:
out = np.zeros(2000,)
for i in range(2000):
    for j in range (100):
        out[i] += X[i,j] * theta[j]
```

Out[12]:
```
(2000, 1)
```

In [11]:
```
import numpy as np

X = np.ones([2000, 100])
Y = np.zeros([2000, 100])
theta = np.random.randn(100,1)
```

$$||x - y||_2^2 = (x-y)^\top (x-y) = x^\top x - 2x^\top y + y^\top y$$

for all x in X and all y in Y

In [16]:
```
# fast:
out = X.dot(Y.T)
out.shape
```

Out[16]:
```
(2000, 2000)
```

In [17]:
```
# slow:
out = np.zeros([2000,2000])
for i in range(2000):
    for j in range(2000):
        out[i,j] = X[i].dot(Y[j])
```

In [20]:
```
print(X.shape)
print(theta.T.shape)
print((X-theta.T).shape)
```

```
(2000, 100)
(1, 100)
(2000, 100)
```

In [ ]:
```
for i in range(2000):
    X[i] - theta[i]
```

In [22]:
```
print(X[np.newaxis, :, :].shape)
print(Y[:, np.newaxis, :].shape)
```

```
(1, 2000, 100)
(2000, 1, 100)
```

In [23]:
```python
print((X[np.newaxis, :, :] - Y[:, np.newaxis, :]).shape)
```
```
(2000, 2000, 100)
```

In [24]:
```python
# slow:
out = np.zeros([2000,2000,100])
for i in range(2000):
    for j in range(2000):
        out[i,j,:] = X[i] - Y[j]

# fast
X[np.newaxis, :, :] - Y[:, np.newaxis, :]
```
```
(2000, 2000)
```

In [ ]:
```python

```