Problem Set 5: Probabilistic Models

This assignment requires a working IPython Notebook installation, which you should already have. If not, please refer to the instructions in the previous problem sets.

Only PDF files are accepted for submission. To print this notebook to a pdf file, you can go to "File" -> "Download as" -> "PDF via LaTex(.pdf)" or simply use "print" in browser.

Total: 100 points (+20 points bonus).

1 Linear Discriminant Analysis (LDA) [50 pts + 10pts bonus]

In this exercise, we build a linear discriminant analysis (LDA) classifier for the problem of predicting whether a student gets admitted into a university.

LDA is a generative model for classification. Given a training dataset of positive and negative features (x, y) with $y \in 0, 1$, LDA models the data x as generated from class-conditional Gaussians:
$P(x, y) = P(x|y)P(y)$, where $P(y = 1) = \pi$ and $P(x|y = j) = N(x; \mu^j, \Sigma)$. Here means μ^j are class-dependent but the covariance matrix Σ is class-independent (the same for all classes).

A novel feature x is classified as a positive if $P(y = 1|x) > P(y = 0|x)$, which is equivalent to $a(x) > 0$, where the linear classifier $a(x) = w^T x + w_0$ has weights given by $w = \Sigma^{-1}(\mu^1 - \mu^0)$.

In practice, and in this assignment, we use $a(x) >$ some threshold, or equivalently, $w^T x > T$ for some constant T. The specific threshold can be determined on a validation dataset or via cross-validation.

As we saw in lecture, LDA and logistic regression can be expressed in the same form

$$P(y = 1|x) = \frac{1}{1 + e^{-\theta^T x}}.$$

However, they generally produce different solutions for the parameter theta.

1.1 Derivation of LDA [20 pts]

Show that the log-odds decision function a(x) for LDA

$$a(x) = \ln \frac{p(x|y = 1)p(y = 1)}{p(x|y = 0)p(y = 0)}$$

is linear in x, that is, we can express $a(x) = w^T x + w_0$ for some w, w_0. Show all your steps.

Answer:

We can rewrite:

$$a(x) = \ln \frac{p(x|y = 1)p(y = 1)}{p(x|y = 0)p(y = 0)}$$

as:

$$a(x) = \ln \frac{p(y=1|x)}{p(y=0|x)}$$

we can then simplify that as:

$$a(x) = \ln \frac{f_1(x)}{f_0(x)} + \ln \frac{\pi_1}{\pi_0}$$

$$a(x) = \ln \frac{\pi_1}{\pi_0} - \frac{1}{2}(\mu_1 + \mu_0)^T \Sigma^{-1}(\mu_1 - \mu_0) + x^T \Sigma^{-1}(\mu_1 - \mu_0)$$

setting

$$w_0 = -\frac{1}{2}\mu_T^1 \Sigma^{-1}\mu_1 + \frac{1}{2}\mu_T^0 \Sigma^{-1}\mu_0 + \ln \frac{p(C_1)}{p(C_2)}$$

and

$$w = \Sigma^{-1}(\mu_1 - \mu_0)$$

we can rewrite the expression as:

$$w^T x + w_0$$

Implementation

In this part, you can assume the prior probabilities for the two classes are the same (although the number of the positive and negative samples in the training data is not the same), and that the threshold T is zero (you do not need to find w_0). As a bonus, you are encouraged to explore how the different prior probabilities shift the decision boundary.

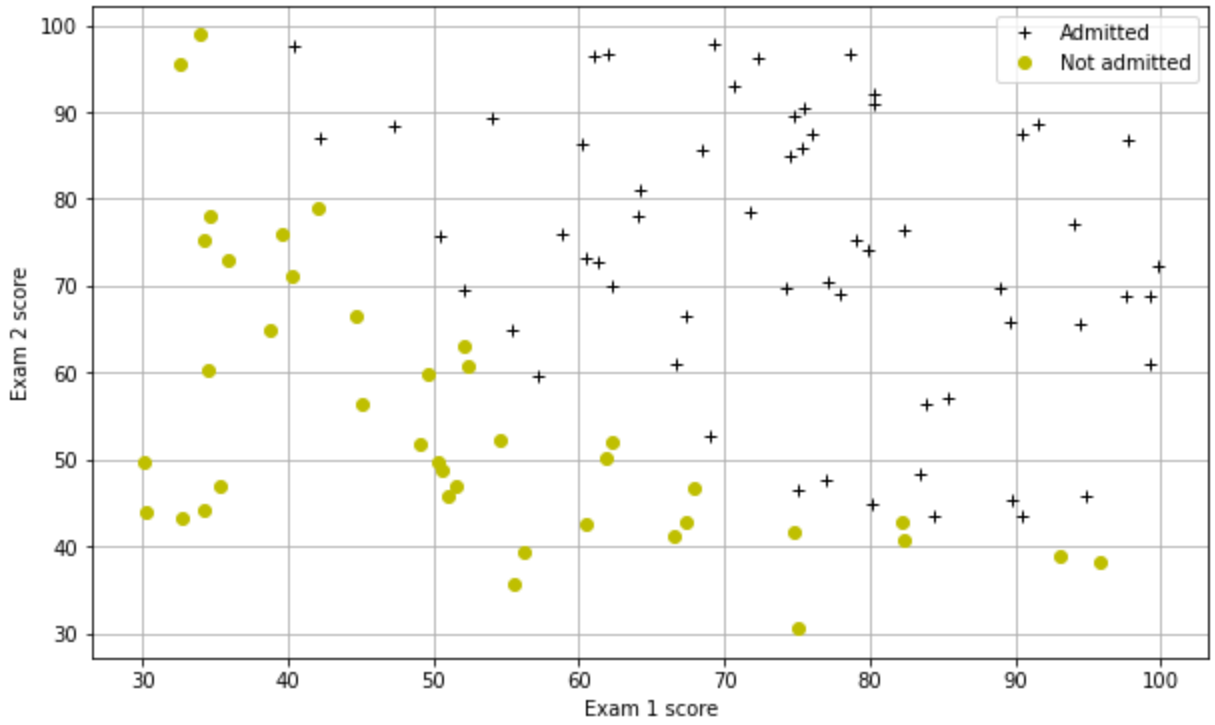
1.2 Loading the data

First, lets load and plot the data.

In [62]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

datafile = 'data/ex2data1.txt'
#!head $datafile
cols = np.loadtxt(datafile, delimiter=',', usecols=(0, 1, 2), unpack=True)  #Read in comma
##Form the usual "X" matrix and "y" vector
X = np.transpose(np.array(cols[:-1]))
y = np.transpose(np.array(cols[-1:]))
m = y.size  # number of training examples
##Insert the usual column of 1's into the "X" matrix
X = np.insert(X, 0, 1, axis=1)

#Divide the sample into two: ones with positive classification, one with null classificati
pos = np.array([X[i] for i in range(X.shape[0]) if y[i] == 1])
neg = np.array([X[i] for i in range(X.shape[0]) if y[i] == 0])

def plotData():
    plt.figure(figsize=(10, 6))
    plt.plot(pos[:, 1], pos[:, 2], 'k+', label='Admitted')
    plt.plot(neg[:, 1], neg[:, 2], 'yo', label='Not admitted')
```

```
    plt.xlabel('Exam 1 score')
    plt.ylabel('Exam 2 score')
    plt.legend()
    plt.grid(True)

plotData()
```



Implement the LDA classifier by completing the code below.

1.3 Centering the data [5 pts]

As an implementation detail, you should first center the positive and negative data separately, so that each set has a mean equal to [0, 0], before computing the covariance, as this tends to give a more accurate estimate of the covariance.

In [88]:
```
# IMPLEMENT THIS
pos_mean = np.mean(pos[:,1:], axis=0)
neg_mean = np.mean(neg[:,1:], axis=0)

middle_value = (pos_mean + neg_mean) / 2

pos_data = pos[:,1:] - middle_value
neg_data = neg[:,1:] - middle_value
```

1.4 Computing parameters and predictions [20 pts]

Implement the LDA algorithm here in vectorized form (avoid loops). First, compute the covariance matrix, then the classifier's weights, then use the classifier to make predictions on the training data. Note, you should center the whole training data set before applying the classifier. Namely, subtract the mean value of the two classes' means (1/2*(pos_mean+neg_mean)), which is on the separating plane when their prior probabilities are the same and becomes the 'center' of the data.

In [89]:
```
# IMPLEMENT THIS
centered_data = np.concatenate((neg_data,pos_data),axis=0)
```

```
cov_all = np.cov(pos_data.T) + np.cov(neg_data.T)
w = np.matmul(np.linalg.inv(cov_all), (pos_mean - neg_mean).T)
y_lda = np.matmul(centered_data, w)
y_lda[y_lda >= 0] = 1
y_lda[y_lda < 0] = 0

print('shape:', cov_all.shape)
print('w:', w)
print('shape:', y_lda.shape)
```

```
shape: (2, 2)
w: [0.0785182  0.07571361]
shape: (100,)
```

1.5 Training accuracy [5 pts]

Complete the code to compute the training set accuracy, which should be around 89%.

In [65]:

```
# IMPLEMENT THIS
accuracy = np.sum(y_lda == sorted(y.flat, reverse=True)) / y.shape[0]
print(accuracy)
```

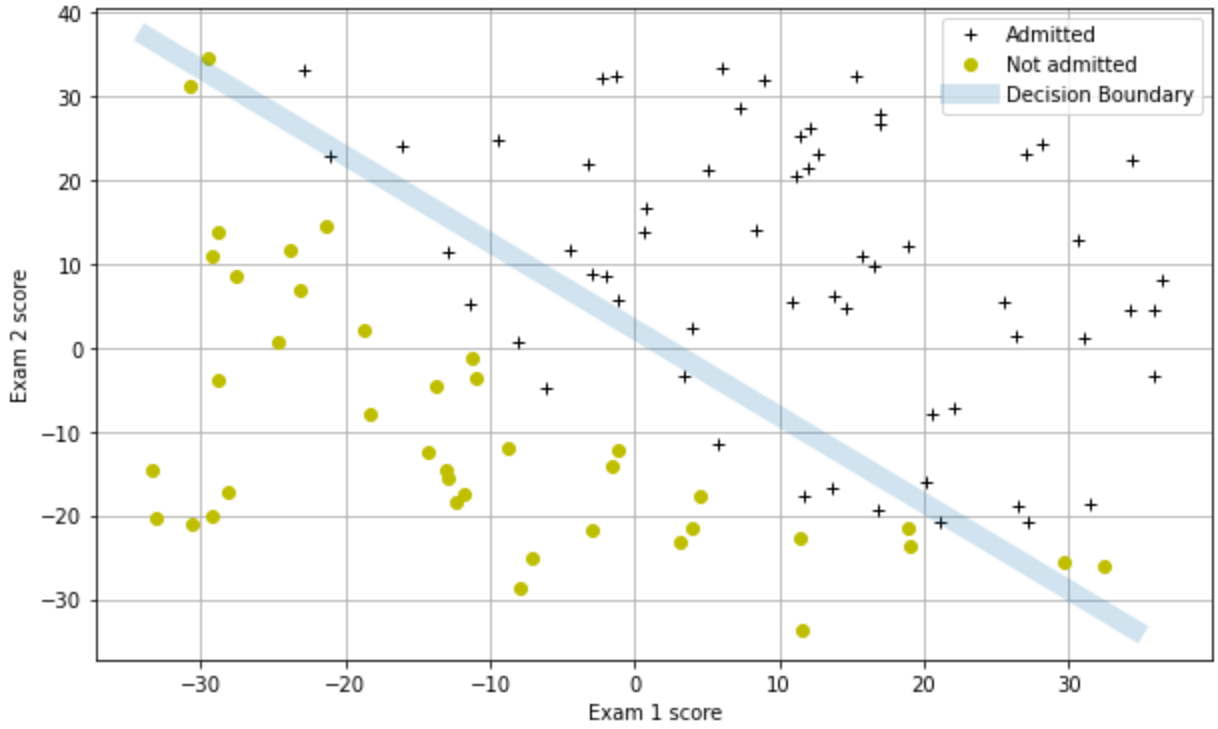
```
0.89
```

1.6 Bonus [10 pts] (optional)

As an optional bonus exercise, try changing the prior probability of the positive class $P(y = 1) = \pi$ and show how the resulting decision boundary changes by plotting it. What is the effect of changing π on the boundary?

In [100…

```
# IMPLEMENT THIS
plt.figure(figsize=(10, 6))
plt.plot(pos_data[:, 0], pos_data[:, 1], 'k+', label='Admitted')
plt.plot(neg_data[:, 0], neg_data[:, 1], 'yo', label='Not admitted')
plt.xlabel('Exam 1 score')
plt.ylabel('Exam 2 score')
plt.legend()
plt.grid(True)

linexs = np.array([np.max(centered_data[:,1]),np.min(centered_data[:,])])
alpha= 0.9
lineys = np.log(alpha/(1-alpha))-linexs*w[0]/w[1]

plt.plot(linexs, lineys, lw=10, alpha=0.2, label='Decision Boundary')
plt.legend()
plt.show()
```



2 Conjugate Prior [25 pts]

In this problem we will estimate the parameter θ of the coin flipping problem using the Bayesian approach. In the Bayesian approach, θ is considered a random variable sampled from some prior distribution $p(\theta)$. We represent the ith coin flip by a Bernoulli random variable $x \in \{0, 1\}$, where $x_i = 1$ if the ith flip lands on heads. We assume that the observed x_i's are conditionally indendent given θ. This means that the joint distribution of n coin flips and θ can be factorized as

$$p(x_1, \ldots, x_n, \theta) = p(\theta) \prod_{i=1}^{n} p(x_i|\theta)$$

Suppose that our prior distribution on θ is $Beta(h, t)$, for some $h, t > 0$. That is,

$$p(\theta) \propto \theta^{h-1}(1-\theta)^{t-1}$$

See Bishop Ch 2.1.1 or https://en.wikipedia.org/wiki/Beta_distribution for more details on this distribution. There is a nice online graphing tool here http://eurekastatistics.com/beta-distribution-pdf-grapher/.

The Beta distribution is a conjugate to the Bernoulli distribution since the prior $p(\theta)$ and posterior $p(\theta|x_1, \ldots, x_n)$ are in the same family (the Beta family).

2.1 Posterior [10 pts]

Suppose our dataset of flips has H heads and T tails. Show that the posterior distribution fo θ is $Beta(h + H, t + T)$, i.e. show that

$$p(\theta|x_1, \ldots, x_n) \propto \theta^{h-1+H}(1-\theta)^{t-1+T}$$

Answer:

Using bayes rule, the formula for the posterior is (Posterior = Prior x Likelihood):

$$p(\theta|x_1, \ldots, x_n) = p(\theta)p(x_1, \ldots, x_n|\theta)$$

We know the maximum likelihood estimate is:

$$p(x_1, \ldots, x_n|\theta) = \theta^{|T|}(1-\theta)^{|H|}$$

We know the prior is a Beta distribution that takes the form:

$$p(\theta) = p(\theta|h,t) = \frac{\Gamma(h+t)}{\Gamma(h)\Gamma(t)}\theta^{h-1}(1-\theta)^{t-1}$$

Now plugging that into bayes rule we get:

$$p(\theta|x_1, \ldots, x_n) \propto \theta^{h-1+H}(1-\theta)^{t-1+T}$$

2.2 Parameter estimates [10 pts]

Give the expressions for three estimates of θ: the Maximum Likelihood (θ_{ML}), the Maximum a Posteriori (θ_{MAP}) or the mode of the posterior over θ, and the mean of the posterior (θ_{MP}).

Hint: You may use the fact that a Beta(h, t) distribution has mean $h/(h+t)$ and has mode $(h-1)/(h+t-2)$ for $h, t > 1$.

Answer:

Taking the log of $p(\theta|x_1, \ldots, x_n) \propto \theta^{h-1+H}(1-\theta)^{t-1+T}$ we get: $\theta_{MAP} = \frac{H+h-1}{H+|T|+h+t-2}$

From the MLE $p(x_1, \ldots, x_n|\theta) = \theta^{T}(1-\theta)^{H}$ we can solve for θ --> $\theta_{ML} = \frac{H}{H+T}$

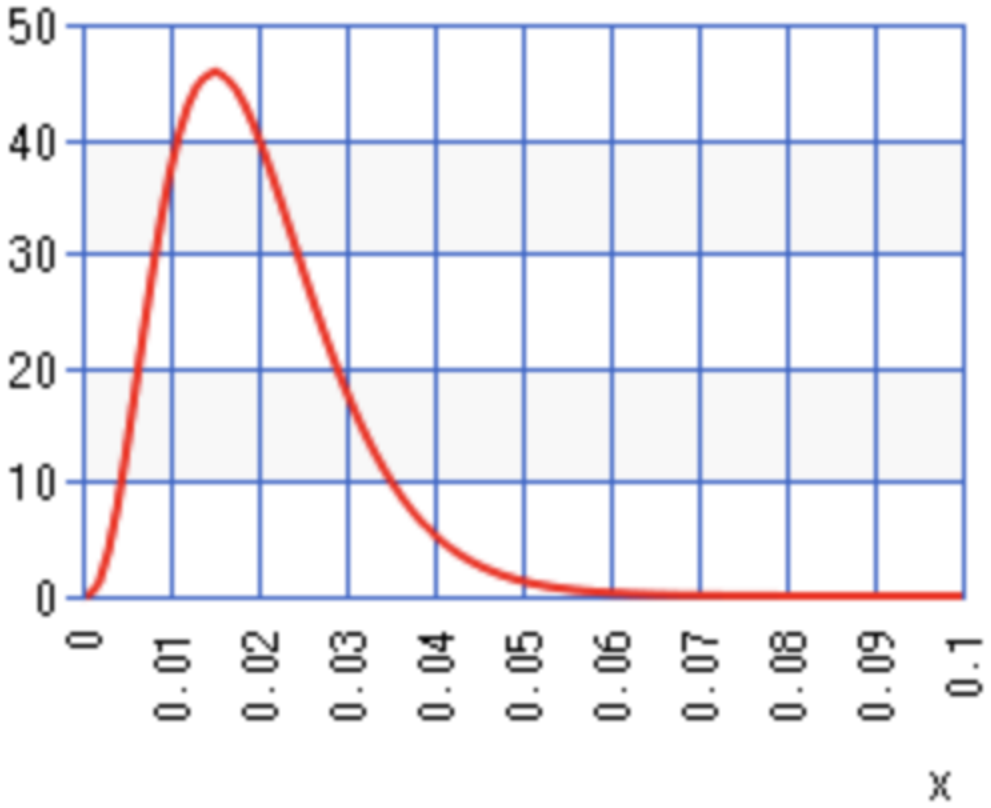
$\theta_{MP} = \frac{H+h}{H+T+h+t}$

2.3 Example [5 pts]

Supose you are modeling some event with a binary outcome (such as a user clicking on an ad after viewing it). You know that typical fractions of positive events observed in data are between 0.01 and 0.02. What are reasonable hyperparameters of the Beta prior to use, and why? Include a plot of the resulting prior PDF.

Answer:

A reasonable hyperparameter to use for Beta prior would be 200. This would make the mean of the prior pdf 0.015. We know that typical fractions of positive events observed in data are between 0.01 and 0.02 so it makes sense for the mean of our prior to be 0.015.



3 Bayesian Linear Regression [25 pts + 10pts bonus]

In this exercise, we implement Bayesian regression with linear basis function models. Recall that a linear regression model $y(\mathbf{x},\mathbf{w})$ can be defined more generally as

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x})$$

where ϕ_j are basis functions and M is the total number of parameters w_j including the bias term w_0. The target variable t of an observation $\mathbf{x}$ is given by a deterministic function $y(\mathbf{x}, \mathbf{w})$ plus additive random noise ϵ.

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon$$

We assume that the noise is normally distributed i.e. follows a Gaussian distribution with zero mean and precision (inverse variance) β. The corresponding probabilistic model i.e. the conditional distribution of t given $\mathbf{x}$ can therefore be written as

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{\beta}{2}(t - y(\mathbf{x}, \mathbf{w}))^2\right)$$

where the mean of this distribution is the regression function $y(\mathbf{x}, \mathbf{w})$.

3.1 Posterior and posterior predictive distribution [10 pts]

For a Bayesian treatment of linear regression, we need a prior probability distribution over the model parameters $\mathbf{w}$ with zero mean:

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|0, \alpha^{-1}\mathbf{I})$$

where $\alpha^{-1}I$ is the diagonal covariance matrix where all diagonal elements have the same variance α^{-1}.

The posterior distribution has the following mean and inverse covariance matrix:

$$m_N = \beta \mathbf{S}_N \Phi^T \mathbf{t}$$

$$\mathbf{S}_N^{-1} = \alpha \mathbf{I} + \beta \Phi^T \Phi$$

Hence, the posterior distribution can be written as

$$p(\mathbf{w}|\mathbf{t}, \alpha, \beta) = \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N)$$

For making a prediction t at a new location $\mathbf{x}$ we use the posterior predictive distribution which is defined as

$$p(t|\mathbf{t}, \mathbf{w}, \alpha, \beta) = \int p(t|\mathbf{x}, \mathbf{w}, \beta) p(\mathbf{w}|\mathbf{t}, \alpha, \beta) d\mathbf{w}$$

Implement the posterior and posterior predictive distributions.

In [1]:

```
import numpy as np

def posterior(Phi, t, alpha, beta, return_inverse=False):
    """Computes mean and covariance matrix of the posterior distribution."""
    S_N_inv = alpha * np.eye(Phi.shape[1]) + beta * Phi.T.dot(Phi)
    S_N = np.linalg.inv(S_N_inv)
    m_N = beta * S_N.dot(Phi.T).dot(t)

    if return_inverse:
        return m_N, S_N, S_N_inv
    else:
        return m_N, S_N


def posterior_predictive(Phi_test, m_N, S_N, beta):
    """Computes mean and variances of the posterior predictive distribution."""
    a = Phi_test.dot(m_N)
    b = 1 / beta + np.sum(Phi_test.dot(S_N) * Phi_test, axis=1)

    return a, b
```

3.2 Generating the data [5 pts]

Target values $\mathbf{t}$ are generated from the design matrix $\mathbf{X} \in \mathbb{R}^{N \times 1}$ with a linear function f which can also generate random noise of specified variance. Implement f below.

In [43]:

```
w0 = -0.3
w1 =  0.5


def f(X, noise_variance):
    return w0 + w1 * X + noise(X.shape, noise_variance)

def noise(size, variance):
    return np.random.normal(scale=np.sqrt(variance), size=size)
```

3.3 Basis functions [5 pts]

For straight line fitting, we do not need to transform x with a basis function, which is equivalent to using an identity basis function. Other basis functions that are non-linear are necessary to model the non-linear relationship between input x and target t. Below is an example of non-linear basis functions: the polynomial. Implement the polynomial basis function. *Note: You will not use it in the rest of the exercise but your implementation should be reasonable.*

In [44]:
```
def identity_basis_function(x):
    return x


def polynomial_basis_function(x, power):
    return x ** power


def expand(x, bf, bf_args=None):
    if bf_args is None:
        return np.concatenate([np.ones(x.shape), bf(x)], axis=1)
    else:
        return np.concatenate([np.ones(x.shape)] + [bf(x, bf_arg) for bf_arg in bf_args],
```

3.3 Straight line fitting [5 pts]

For straight line fitting, we use a linear regression model of the form $y(\mathbf{w}, \mathbf{x}) = w_0 + w_1 x$ and perform Bayesian inference for model parameters $\mathbf{w}$. Predictions are made with the posterior predictive distribution. Since this model has only two parameters, w_0 and w_1, we can visualize the posterior density in 2D which is done in the first column of the following output. Rows use an increasing number of training data from a training dataset.

How does the dataset size affect the posterior density in the first column of the plots?

Answer:

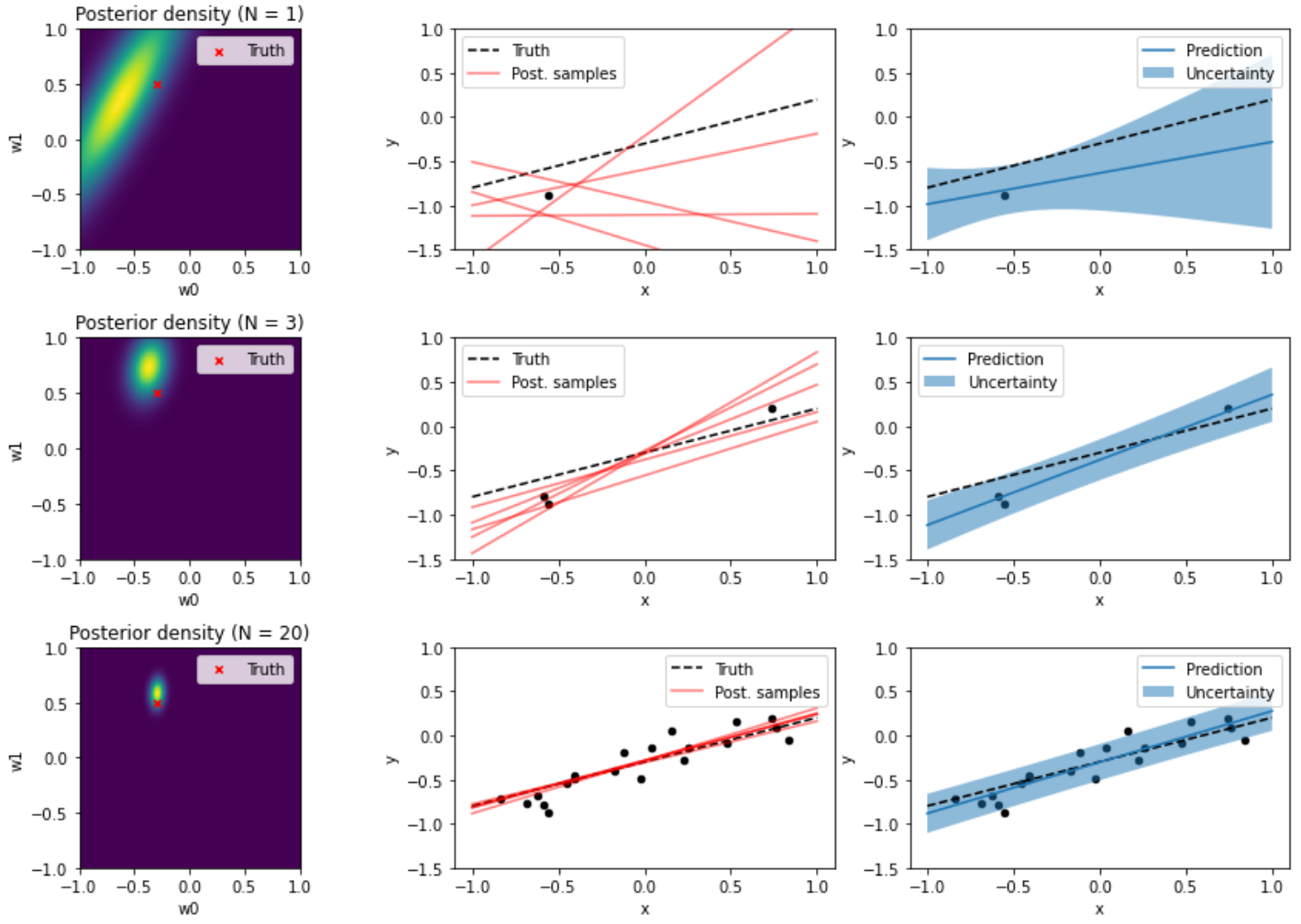
In [45]:
```
from scipy import stats
import matplotlib.pyplot as plt

def plot_data(x, t):
    plt.scatter(x, t, marker='o', c="k", s=20)


def plot_truth(x, y, label='Truth'):
    plt.plot(x, y, 'k--', label=label)


def plot_predictive(x, y, std, y_label='Prediction', std_label='Uncertainty', plot_xy_labe
    y = y.ravel()
    std = std.ravel()

    plt.plot(x, y, label=y_label)
    plt.fill_between(x.ravel(), y + std, y - std, alpha = 0.5, label=std_label)

    if plot_xy_labels:
        plt.xlabel('x')
        plt.ylabel('y')


def plot_posterior_samples(x, ys, plot_xy_labels=True):
    plt.plot(x, ys[:, 0], 'r-', alpha=0.5, label='Post. samples')
    for i in range(1, ys.shape[1]):
        plt.plot(x, ys[:, i], 'r-', alpha=0.5)
```

```
    if plot_xy_labels:
        plt.xlabel('x')
        plt.ylabel('y')


def plot_posterior(mean, cov, w0, w1):
    resolution = 100

    grid_x = grid_y = np.linspace(-1, 1, resolution)
    grid_flat = np.dstack(np.meshgrid(grid_x, grid_y)).reshape(-1, 2)

    densities = stats.multivariate_normal.pdf(grid_flat, mean=mean.ravel(), cov=cov).resha
    plt.imshow(densities, origin='lower', extent=(-1, 1, -1, 1))
    plt.scatter(w0, w1, marker='x', c="r", s=20, label='Truth')

    plt.xlabel('w0')
    plt.ylabel('w1')


def print_comparison(title, a, b, a_prefix='np', b_prefix='br'):
    print(title)
    print('-' * len(title))
    print(f'{a_prefix}:', a)
    print(f'{b_prefix}:', b)
    print()
```

In [46]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# fix random seed so we always get the same data
np.random.seed(5)

# Training dataset sizes
N_list = [1, 3, 20]

beta = 25.0
alpha = 2.0

# Training observations in [-1, 1)
X = np.random.rand(N_list[-1], 1) * 2 - 1

# Training target values
t = f(X, noise_variance=1/beta)

# Test observations
X_test = np.linspace(-1, 1, 100).reshape(-1, 1)

# Function values without noise
y_true = f(X_test, noise_variance=0)

# Design matrix of test observations
Phi_test = expand(X_test, identity_basis_function)

plt.figure(figsize=(15, 10))
plt.subplots_adjust(hspace=0.4)

for i, N in enumerate(N_list):
    X_N = X[:N]
    t_N = t[:N]

    # Design matrix of training observations
    Phi_N = expand(X_N, identity_basis_function)
```

```
    # Mean and covariance matrix of posterior
    m_N, S_N = posterior(Phi_N, t_N, alpha, beta)

    # Mean and variances of posterior predictive
    y, y_var = posterior_predictive(Phi_test, m_N, S_N, beta)

    # Draw 5 random weight samples from posterior and compute y values
    w_samples = np.random.multivariate_normal(m_N.ravel(), S_N, 5).T
    y_samples = Phi_test.dot(w_samples)

    plt.subplot(len(N_list), 3, i * 3 + 1)
    plot_posterior(m_N, S_N, w0, w1)
    plt.title(f'Posterior density (N = {N})')
    plt.legend()

    plt.subplot(len(N_list), 3, i * 3 + 2)
    plot_data(X_N, t_N)
    plot_truth(X_test, y_true)
    plot_posterior_samples(X_test, y_samples)
    plt.ylim(-1.5, 1.0)
    plt.legend()

    plt.subplot(len(N_list), 3, i * 3 + 3)
    plot_data(X_N, t_N)
    plot_truth(X_test, y_true, label=None)
    plot_predictive(X_test, y, np.sqrt(y_var))
    plt.ylim(-1.5, 1.0)
    plt.legend()
```



3.4 Bonus [10 pts] (optional)

We now fit a Gaussian basis function model to a noisy sinusoidal dataset for which we provide a sinusoidal function g with noise variance. Implement the gaussian basis function below.

The model uses 7 Gaussian basis functions with mean values equally distributed over [0,1] each having a standard deviation of 0.1. We then infer the model parameters **w** using the posterior predictive distribution.

How many parameters w_i are there? Run Bayesian inference and generate the plots. Do they follow the same trend as the ones generated by Bayesian linear regression?

Answer:

In [47]:
```
def g(X, noise_variance):
    '''Sinusoidial function plus noise'''
    return 0.5 + np.sin(2 * np.pi * X) + noise(X.shape, noise_variance)


def gaussian_basis_function(x, mu, sigma=0.1):
    return np.exp(-0.5 * (x - mu) ** 2 / sigma ** 2)
```

In [48]:
```
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import curve_fit

# fix random seed so we always get the same data
np.random.seed(5)

N_list = [3, 8, 20]

beta = 25.0
alpha = 2.0

# Training observations in [-1, 1)
X = np.random.rand(N_list[-1], 1)

# Training target values
t = g(X, noise_variance=1/beta)

# Test observations
X_test = np.linspace(0, 1, 100).reshape(-1, 1)

# Function values without noise
y_true = g(X_test, noise_variance=0)

# Design matrix of test observations
Phi_test = expand(X_test, bf=gaussian_basis_function, bf_args=np.linspace(0, 1, 7))

plt.figure(figsize=(10, 10))
plt.subplots_adjust(hspace=0.4)

for i, N in enumerate(N_list):
    X_N = X[:N]
    t_N = t[:N]

    # Design matrix of training observations
    Phi_N = expand(X_N, bf=gaussian_basis_function, bf_args=np.linspace(0, 1, 7))

    # Mean and covariance matrix of posterior
    m_N, S_N = posterior(Phi_N, t_N, alpha, beta)

    # Mean and variances of posterior predictive
    y, y_var = posterior_predictive(Phi_test, m_N, S_N, beta)
```

```
    # Draw 5 random weight samples from posterior and compute y values
    w_samples = np.random.multivariate_normal(m_N.ravel(), S_N, 5).T
    y_samples = Phi_test.dot(w_samples)

    plt.subplot(len(N_list), 2, i * 2 + 1)
    plot_data(X_N, t_N)
    plot_truth(X_test, y_true)
    plot_posterior_samples(X_test, y_samples)
    plt.ylim(-1.0, 2.0)
    plt.legend()

    plt.subplot(len(N_list), 2, i * 2 + 2)
    plot_data(X_N, t_N)
    plot_truth(X_test, y_true, label=None)
    plot_predictive(X_test, y, np.sqrt(y_var))
    plt.ylim(-1.0, 2.0)
    plt.legend()
```