CS 181 Spring 2022 Section 2 Notes: Probabilistic Regression, Classification

1 Probabilistic Regression (Review)

The idea behind probabilistic regression is to assume that there is a "story" for how the data were created. For a model parameterized by θ, the **likelihood** of the data $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$, $\mathbf{x}_i \in \mathbb{R}^m$, $y_i \in \mathbb{R}$ appearing, given the specific parameter θ is defined as:

$$L(\theta|D) = p(D|\theta), \text{ which is often also written as } f(D|\theta).$$

1. Note that θ may contain *multiple* elements!
2. The value of θ that maximizes the likelihood is called the maximum likelihood estimate or **MLE**. We find the MLE by taking the derivative of the likelihood (or log-likelihood, see below), setting it equal to 0, and solving for θ. Rigorously, we also should verify the second derivative to be negative, but in practice, this may be occasionally omitted.
3. Note that in other courses like Stat 111, we might write $L(\theta; D)$, instead of $L(\theta|D)$. They mean exactly the same thing.
4. For those of you who have taken Stat 110 and/or Stat 111, $f(D|\theta)$ is also known as the "joint PDF" of the data. However, in statistics and machine learning, there is a slight difference between the joint PDF $f(D|\theta)$ and the likelihood function $L(\theta|D)$, *even though they are mathematically identical*: the likelihood function is interpreted as a function of the parameter(s) θ, while the joint PDF is interpreted as a function of the observed data.
5. If we want, we can further add a generative story for θ (and make it random). This is called the **prior distribution**, or just **prior** of θ: $p(\theta)$

Now, onto probabilistic regression itself:

1. For probabilistic regression, we modeled the conditional distribution, $p(y|\mathbf{x})$, with a target value y_i Normally distributed with mean $\mathbf{w}^\top \mathbf{x}_i$ and variance σ^2.
2. We showed that finding parameters $\mathbf{w}$ that minimizes the negative log-likelihood (see below for explanation) of labels $\mathbf{y}$ given design matrix $\mathbf{X}$ gives the same expression for the optimal parameters $\mathbf{w}^*$ as from using ordinary least squares regression.
3. Later we will also see a "full Bayes" approach where we also reason about priors on the parameters θ.

A quick addendum: what (else) can we do with our generative model?

1. Based on the generative model, and using Bayes' rule, we can find the **posterior distribution** for θ
$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)} \propto p(D|\theta)p(\theta)$$

Note that $L(\theta|D)$ *is NOT* the same thing as $p(\theta|D)$. This may seem confusing, but just remember that $L(\theta|D) = p(D|\theta)$! So, in other words, we have the following:

$$p(\theta|D) \propto L(\theta|D)p(\theta)$$

Again, this is confusing! Please read this line again. Some of you may know this distinction as the Bayesian vs. Frequentist schools of statistics.

2. The value θ that maximizes the **posterior distribution** is called the maximum aposteriori or **MAP** estimate.

3. A good way to remember the **posterior distribution** is the saying: "posterior is *proportional* to likelihood times prior."

Remarks on Taking the Log:

1. When we are maximizing the likelihood or the posterior, often, we apply $\log$ on both sides of the equation so that we are then maximizing the **log-likelihood** or **log-posterior**.

2. Because $\log$ is a monotonically-increasing function, the value of θ that maximizes the log-likelihood (or log-posterior) also maximizes the original likelihood (or posterior, if we're working with the posterior).

3. Practically, this is helpful because the log operation turns products into sums, which are easier to take the derivative of (because the product rule is not the most pleasant thing to do).

4. The **log-likelihood function** will often be denoted $\ell(\theta|D)$

2 Linear Classification

2.1 Takeaways

2.1.1 Classification

1. Goal : Given an input vector $\mathbf{x}$, assign it to one of K discrete classes C_k.

2. Strategy: Divide our input space into *disjoint* (i.e., no overlap) **decision regions** whose boundaries are called **decision boundaries** or **decision surfaces**.

3. Note: each decision region corresponds to being assigned to a certain class: there should be K decision regions if we are working with K discrete classes.

2.1.2 Binary Linear Classification

- We are working with two classes divided by a linear separator in our feature space. We will denote the two classes as -1 and 1 (note that in other situations, we might use 0 and 1).

- Note that linear in this sense is *not* limited to the 2D case. Formally, if each data point has D dimensions, then the linear separator dividing our two classes (also called a "hyperplane") has $D - 1$ dimensions. For example, if each data point has 3 dimensions, then the linear separator / hyperplane is a 2D plane.

- **Discriminant function** : Function that directly assigns each vector to a specific class

$$\hat{y} = \text{sign}(h(\mathbf{x}; \mathbf{w}, w_0)) = \text{sign}(\mathbf{w}^\top \mathbf{x} + w_0)$$

  *note: $sign(z) = 1$ if $z \geq 0$, and $sign(z) = -1$ if $z < 0$.

- $\mathbf{w}$ is orthogonal to every point on the decision surface. It determines the orientation of the decision boundary.

2.1.3 Perceptron

- Perceptron is a discriminative algorithm for binary classification that finds a linear decision boundary surface, if one exists.
- To define the loss, we use the hinge loss / rectified linear function, also called $ReLU$:

$$ReLU(z) = \max\{0, z\}$$

- We define the Perceptron loss function as follows, with $h(\mathbf{x}_i; \mathbf{w}, w_0) = \mathbf{w}^\top \mathbf{x}_i + w_0$:

$$\begin{aligned}\mathcal{L}(\mathbf{w}) &= \sum_{i=1}^{n} ReLU(-h(\mathbf{x}_i; \mathbf{w}, w_0)y_i) \\ &= -\sum_{i=1: y_i \neq \hat{y}_i}^{n} (\mathbf{w}^\top \mathbf{x}_i + w_0)y_i\end{aligned}$$

- We can find our optimal weights by updating using (stochastic) gradient descent. Below is the equation for updating the weights $\mathbf{w}$ at time t using the i^{th} data point. Note that Perceptron updates its weights one data point at a time.

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\partial}{\partial \mathbf{w}} \mathcal{L}^{(i)}(\mathbf{w}) = \mathbf{w}^{(t)} + \eta y_i \mathbf{x}_i,$$

2.2 Concept Question

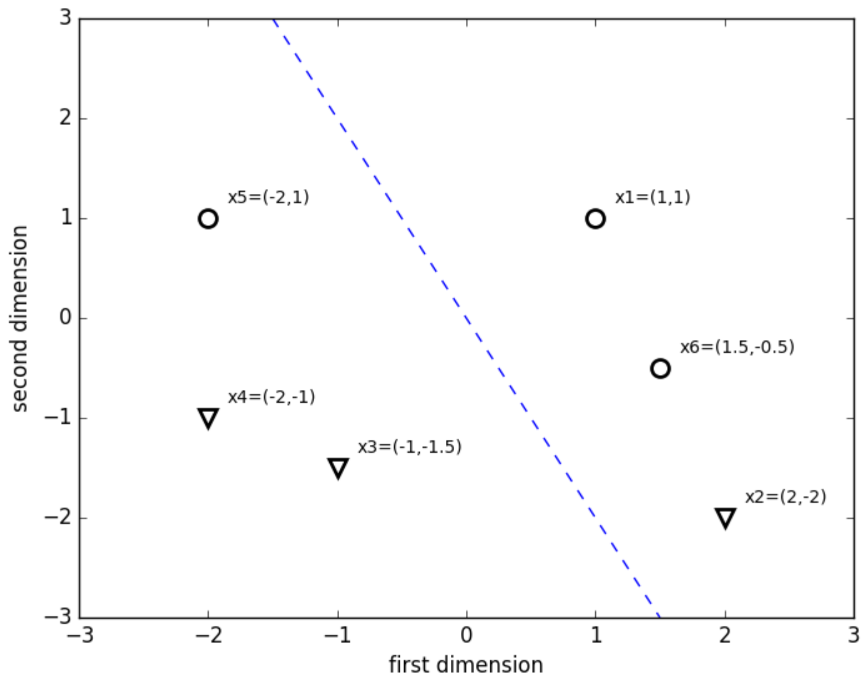
Why do we choose the $ReLU$ function over the $0/1$ function when formulating the loss function?

2.3 Exercise: Small Perceptron Example

Let's train a perceptron on a small data set. Consider data $\{\mathbf{x}_i\}_{i=1}^{N}, \mathbf{x}_i \in \mathbb{R}^2$. Let the learning rate $\eta = 0.2$ and let the weights be initialized as:

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.5 \end{pmatrix}, w_0 = 0$$

Let the circles have $y_i = 1$ and the triangles $y_i = -1$. The data and initial separation boundary (determined by $\mathbf{w}$) is illustrated below.



Proceed by iterating over each example until there are no more classification errors. When in doubt, refer to the notes above. We know a priori that we will be able to train the classifier and have no classification errors because one can see visually that the data is linearly separable (note: as mentioned above, if the data were not so obviously linearly separable, a new basis could make it so). How many updates do you have to make? Is this surprising?

3 Probabilistic Classification

3.1 Takeaways

3.1.1 Probabilistic Discriminative Model

1. In general, our goal with probabilistic discriminative modeling is to model $p(y|\mathbf{x})$.
2. Intuitively, and importantly, this means that we do not care about how $\mathbf{x}$ is generated – we just care about the following: *given* $\mathbf{x}$, what is the distribution of y?

A specific type of probabilistic discriminative modeling in the *binary case* is **logistic regression**.

3.1.2 Logistic Regression

1. In binary logistic regression, we only have *two* classes, which we will denote as 0 and 1. Note that we are *not* using -1 and 1 anymore!
2. We will model our probability distribution for the label of a certain data point y, given its features $\mathbf{x}$, as follows, for some weights $\mathbf{w}$ and intercept w_0:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0)$$
$$p(y = 0|\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\mathbf{x} + w_0)$$

   *In some texts, we might just see $\mathbf{w}^T\mathbf{x}$ instead of $\mathbf{w}^T\mathbf{x} + w_0$, because of the bias trick. They mean the same thing.

3. The σ denotes the **sigmoid function**, which is defined as:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

   The **sigmoid function** is important because it takes any value z on the real line (i.e., $\mathbb{R}$) and returns an output on $(0, 1)$. This is very important because probabilities must be between 0 and 1. For clarity, $\exp(-z) = e^{-z}$.

4. To find the best weights $\mathbf{w}$, we need to set up a loss function. We will use what is called the **negative log-likelihood** loss function.
   (a) Intuitively, we want to find $\mathbf{w}$ that maximizes the *likelihood* of our data.
   (b) However, unlike ordinary least squares linear regression from last week, there is no clean-cut analytical solution. Thus, we have to use gradient descent.
   (c) The problem is – gradient descent is used to minimize a function. Well, minimizing the negative likelihood (i.e., likelihood $\times$ -1) is the same thing as maximizing the likelihood. Furthermore, we know that minimizing the log-likelihood is the same thing as minimizing the likelihood, but more mathematically tractable.
   (d) By definition, a "loss function" is something that we want to minimize when trying to find our optimal weights $\mathbf{w}$. Thus, we use the negative log-likelihood as our loss function.

(e) In practice, we will use an iterative method like (stochastic) gradient descent to minimize our negative log-likelihood loss and obtain our optimal weights $\mathbf{w}$.

(f) With a training data set of N points of the form $(\mathbf{x_i}, y_i)$, our negative log-likelihood loss (which, fun fact, is also sometimes called the "cross-entropy loss") is defined as follows:

$$\mathcal{L}(\theta) = -\sum_{n=1}^{N} (y_n \ln p(y_n = 1|\mathbf{x}_n; \theta) + (1 - y_n) \ln p(y_n = 0|\mathbf{x}_n; \theta))$$

(g) After fitting our model, if we want to predict the class y^* for a new data point $\mathbf{x}^*$, we will calculate the following class probabilities, and assign this new data point to which ever class has the higher probability.

$$p(y^* = 1|\mathbf{x}^*; \mathbf{w})$$

$$p(y^* = 0|\mathbf{x}^*; \mathbf{w}) = 1 - p(y^* = 1|\mathbf{x}^*; \mathbf{w})$$

(h) Because of the $\mathbf{w}^T\mathbf{x}$, logistic regression has *linear* decision boundaries! Yes, the sigmoid function isn't a straight line/hyperplane, but the $\mathbf{w}^T\mathbf{x}$ ensures that we have a linear decision boundary!

(i) Remarks on Notation:

i. In some texts, you may see a $\hat{y_i}$ term. Don't be scared! In the context of logistic regression,

$$\hat{y_i} = p(y_i = 1|\mathbf{x_i}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0), \text{ or with the bias trick, just } \sigma(\mathbf{w}^T\mathbf{x}).$$

ii. Instead of $y_i = 1$, in some course materials, you might sometimes see $y_i = C_1$. They mean literally the same thing. Analogously, C_2 corresponds to class 0.

iii. In the expression for the negative log-likelihood loss above, θ refers to the parameters of our model. In the context of logistic regression, θ and $\mathbf{w}$ (with maybe w_0) mean the same thing.

iv. $\mathcal{L}$ refers to the **loss function**, while L refers to the likelihood, and ℓ refers to the log-likelihood. Be sure to check the context in which these symbols are used!

(j) We can also extend logistic regression to the multi-class case using the softmax function. See pg. 42 in *Undergraduate Fundamentals of Machine Learning* for a deeper treatment of this extension.

3.1.3 Generative Model

While a **discriminative model** works with the *conditional distribution* of $p(y|\mathbf{x})$, a **generative model** models the entire *joint distribution* of $p(\mathbf{x}, y)$. By Bayes' Rule, we know that

$$p(\mathbf{x}, y) \propto p(\mathbf{x}|y)p(y)$$

- $p(y)$ is called the **class prior** and is almost always a **categorical distribution**, and is usually just a Bernoulli distribution in the case of binary classification (classes 0 and 1, in this context).
- A **categorical distribution** (Cat) is a generalization of the Bernoulli distribution.
  1. If X is a categorical random variable, we write $X \sim Cat(\mathbf{x}, \boldsymbol{\pi})$ with parameters $\mathbf{x} = [x_1, \ldots, x_k]$ and $\boldsymbol{\pi} = [\pi_1, \ldots, \pi_k]$.
  2. $\mathbf{x}$ is a vector of all the possible values that X can take on.
  3. $\boldsymbol{\pi}$ stores the probabilities of X taking on a particular value.
  4. Of course, all the elements in $\boldsymbol{\pi}$ must sum up to 1 and be nonnegative, because X must take on one of these values and probabilities are always nonnegative.
  5. Mathematically, we write:
  $$P(X = x_i) = \pi_i$$
- The **class prior** (often also known as the "prior distribution of y") $p(y)$ gives an a priori probability of an observation being a certain class. Intuitively, this is our initial belief of the distribution of y before we observe any data.
- $p(\mathbf{x}|y)$ is called the **class-conditional distribution** and its form is model-specific. Intuitively, this tells us given y (our class assignment), how likely we are to see the corresponding $\mathbf{x}$ features.
- We are interested in picking the class k that maximizes $p(y = k|\mathbf{x})$. Again, the following equation (Bayes' Rule) might be helpful:
  $$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \propto p(\mathbf{x}|y)p(y)$$

  *note: depending on the model, $\mathbf{x}$ can be either discrete or continuous. However, y must be discrete, by definition of "classes."

3.1.4 Naive Bayes

Naive Bayes is one type of generative model for classification. It's "naive" because we assume that each dimension $d \in \{1, ..., D\}$ of the nth observed data point $\mathbf{x}_n$ is **conditionally independent** from the other dimensions given the correct class label i.e. $y_n = C_k$.

$$p(\mathbf{x}_n | y_n = C_k) = \prod_{d=1}^{D} p(x_{nd} | y_n = C_k)$$

*note: x_{nd} refers to the d^{th} entry of the n^{th} data point $\mathbf{x_n}$. In Naive Bayes, we assume that each of $x_{n1}, \ldots, x_{nD}$ has some conditional distribution given y_n. Each of these element-wise conditional distributions is independent of each other, by definition of Naive Bayes.

For sake of brevity, please see pg. 47 in *Undergraduate Fundamentals of Machine Learning* for a deeper treatment of Naive Bayes, and a toy example.

3.1.5 Naive Bayes Concept Questions

How many parameters does this model have? Why do we use the "naive" assumption?

4 Additional Exercises

4.1 Exercise: Shapes of Decision Boundaries I

Consider now a generative model with $K > 2$ classes, and output label $\mathbf{y}$ encoded as a "one hot" vector of length K. We adopt class prior $p(\mathbf{y} = C_k; \boldsymbol{\pi}) = \pi_k$ for all $k \in \{1, \ldots, K\}$ (where π_k is a parameter of the prior). Let $p(\mathbf{x} \,|\, \mathbf{y} = C_k)$ denote the class-conditional density of features $\mathbf{x}$ (in this case for class C_k). Let the class-conditional probabilities be Gaussian distributions

$$p(\mathbf{x} \,|\, \mathbf{y} = C_k) = \mathcal{N}(\mathbf{x} \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \text{ for } k \in \{1, \ldots, K\}$$

We will predict the class of a new example $\mathbf{x}$ as the class with the highest conditional probability, $p(\mathbf{y} = C_k \,|\, \mathbf{x})$. Luckily, a little bird came to the window of your dorm, and claimed that you can classify an example $\mathbf{x}$ by finding the class that maximizes the following function:

$$f_k(\mathbf{x}) = \log(\pi_k) - \frac{1}{2}\log(|\boldsymbol{\Sigma}_k|) - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k).$$

Derive this formula by comparing two different classes' conditional probabilities. What can we claim about the shape of the decision boundary given this formula?

4.2 Exercise: Shapes of Decision Boundaries II

Let's say the little bird comes back and now tells you that every class has the same covariance matrix, and so $\boldsymbol{\Sigma}_\ell = \boldsymbol{\Sigma}_\ell'$ for all classes C_ℓ and $C_{\ell'}$. Simplify this formula down further. What can we claim about the shape of the decision boundaries now?

4.3 OPTIONAL: Visualizing Decision Boundaries (This Looks Cool, But *Not* Required)

If you want to better understand what these decision boundaries look like, we can visualize them! Let's consider two classes and assume x lives in 2 dimensions. We first consider the case in which the two classes have identical covariances, here defined as

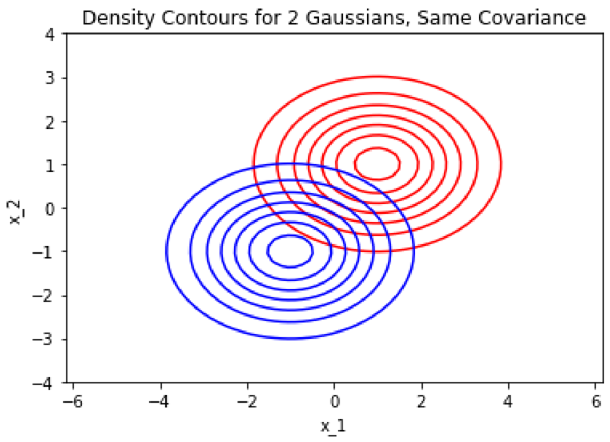
$$p(x|y=1) = \mathcal{N}\Big(\mu_1 = \begin{bmatrix}1\\1\end{bmatrix}, \Sigma_1 = \begin{bmatrix}2 & 0\\0 & 1\end{bmatrix}\Big)$$
$$p(x|y=2) = \mathcal{N}\Big(\mu_2 = \begin{bmatrix}-1\\-1\end{bmatrix}, \Sigma_2 = \begin{bmatrix}2 & 0\\0 & 1\end{bmatrix}\Big)$$

We use the following code to plot the contours of both Gaussians:

```
# imports
import matplotlib.pyplot as plt
import numpy as np
import plotly.graph_objects as go
import scipy.stats

# create meshgrid from -8 to 8
mesh_granularity = 100
possible_vals = np.linspace(-8., 8, mesh_granularity)
mesh_coords = np.meshgrid(possible_vals, possible_vals)
mesh_coords = np.reshape(np.stack(mesh_coords),
                         newshape=(2, mesh_granularity * mesh_granularity)).T

# define means of Gaussians
means = [np.array([1., 1.]),
         np.array([-1., -1.])]

# compute densities for both Gaussians assuming equal covariances
covs = [np.array([[2., 0.],
                  [0., 1.]]),
        np.array([[2., 0.],
                  [0., 1.]])]
same_cov_densities = [scipy.stats.multivariate_normal.pdf(x=mesh_coords,
                                                          mean=mean,
                                                          cov=cov)
                      for mean, cov in zip(means, covs)]

# plot
plt.contour(np.reshape(mesh_coords[:, 0],
                       newshape=(mesh_granularity, mesh_granularity)),
            np.reshape(mesh_coords[:, 1],
                       newshape=(mesh_granularity, mesh_granularity)),
            np.reshape(same_cov_densities[0],
                       newshape=(mesh_granularity, mesh_granularity)),
            colors='red')
plt.contour(np.reshape(mesh_coords[:, 0],
                       newshape=(mesh_granularity, mesh_granularity)),
            np.reshape(mesh_coords[:, 1],
                       newshape=(mesh_granularity, mesh_granularity)),
            np.reshape(same_cov_densities[1],
                       newshape=(mesh_granularity, mesh_granularity)),
            colors='blue')
```

```
plt.xlabel('x_1')
plt.ylabel('x_2')
plt.title('Density Contours for 2 Gaussians, Same Covariance')
plt.axis('equal')
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.show()
```

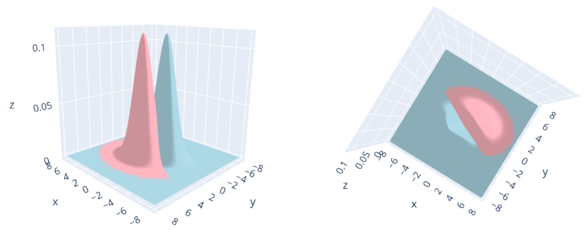
This gives us the following contours:



If you would prefer plotting in 3D, we can alternatively use Plotly:

```
fig = go.Figure(data=[
go.Mesh3d(x=mesh_coords[:, 0],
          y=mesh_coords[:, 1],
          z=same_cov_densities[0],
          color='lightpink'),
go.Mesh3d(x=mesh_coords[:, 0],
          y=mesh_coords[:, 1],
          z=same_cov_densities[1],
          color='lightblue')])
fig.show()
```

Rotating the plot (and ignoring peripheral floating point problems), we can see that the two Gaussians have equal density along a line:

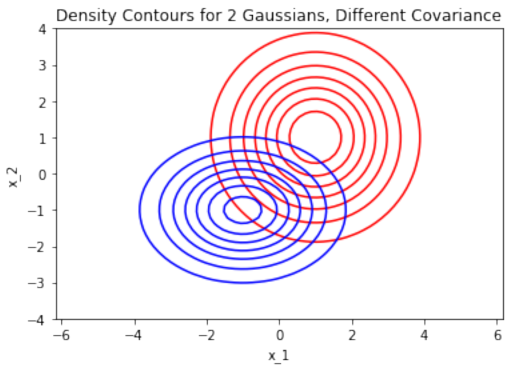


We now plot the second case, with unequal covariances. We assume the two Gaussians are:

$$p(x|y=1) = \mathcal{N}\Big(\mu_1 = \begin{bmatrix}1\\1\end{bmatrix}, \Sigma_1 = \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}\Big)$$
$$p(x|y=2) = \mathcal{N}\Big(\mu_2 = \begin{bmatrix}-1\\-1\end{bmatrix}, \Sigma_2 = \begin{bmatrix}2 & 0\\0 & 1\end{bmatrix}\Big)$$

```
# define means of Gaussians
means = [np.array([1., 1.]),
         np.array([-1., -1.])]

# compute densities for both Gaussians assuming equal covariances
covs = [2.*np.eye(2), np.array([[2., 0.], [0., 1.]])]
diff_cov_densities = [scipy.stats.multivariate_normal.pdf(x=mesh_coords,
                                                          mean=mean,
                                                          cov=cov)
                      for mean, cov in zip(means, covs)]
```

The contour plot outputs as follows:



Rotating the plot (and ignoring the floating point problems on the periphery), we can see that the two Gaussians have equal density along a parabola:

```
fig = go.Figure(data=[
go.Mesh3d(x=mesh_coords[:, 0], y=mesh_coords[:, 1], z=diff_cov_densities[0],
          color='lightpink'),
go.Mesh3d(x=mesh_coords[:, 0], y=mesh_coords[:, 1], z=diff_cov_densities[1],
          color='lightblue')])
fig.show()
```