# Gaussian Process Classification - Detailed Code Analysis

## 1. Model Overview and Architecture

The Gaussian Process Classification (GPC) model implements a non-linear probabilistic binary classifier using:

- A Gaussian Process prior over latent functions
- A Bernoulli likelihood with sigmoid link function
- Laplace approximation for tractable inference

```python
# Hierarchical model structure
y|f(x) ~ Bernoulli(σ(f(x)))  # Observation model
f(x) ~ GP(0, k(x, x'))       # Prior over latent function
```

## 2. Key Components

### 2.1 BernoulliLikelihood Class

This class implements the likelihood p(y|f) = σ(f)^y * (1-σ(f))^(1-y).

python

```python
class BernoulliLikelihood(object):
    def __init__(self, y):
        self.y = y.ravel()  # Store binary labels

    def log_lik(self, f):
        """
        Log likelihood: log p(y|f) = Σ[y_n log σ(f_n) + (1-y_n) log(1-σ(f_n))]
        """
        ll = jnp.sum(self.y * jnp.log(sigmoid(f)) +
                     (1 - self.y) * jnp.log(1 - sigmoid(f)))
        return ll

    def grad(self, f):
        """
        Gradient: ∂log p(y|f)/∂f = y - σ(f)
        """
        g = self.y - sigmoid(f)
        return g

    def hessian(self, f):
        """
        Hessian: ∂²log p(y|f)/∂f² = -diag(σ(f)(1-σ(f)))
        """
        hess_diag = -(sigmoid(f) * (1 - sigmoid(f)))
        Lambda = jnp.diag(hess_diag)
        return Lambda
```

## 2.2 GaussianProcessClassification Class

The main class that implements the full GPC model:

python

```python
class GaussianProcessClassification(object):
    def __init__(self, X, y, likelihood, kernel, kappa=1., lengthscale=1., jitter=1e-8):
        self.X = X                          # Training inputs (N x D)
        self.y = y                          # Training labels (N x 1)
        self.N = len(X)                     # Number of data points
        self.likelihood = likelihood(y)      # Instantiate likelihood
        self.kernel = kernel                # Kernel instance
        self.jitter = jitter               # Numerical stability

        # Set hyperparameters
        self.set_hyperparameters(kappa, lengthscale)

        # Precompute kernel matrix and Cholesky decomposition
        self.K = self.kernel.construct_kernel(self.X, self.X, jitter=self.jitter)
        self.L = jnp.linalg.cholesky(self.K)

        # Construct Laplace approximation
        self.construct_laplace_approximation()
```

## 3. Laplace Approximation Implementation

### 3.1 MAP Estimation

The MAP estimate maximizes the log joint distribution:

```
python
```

```
def log_joint_a(self, a):
    """
    Log joint with reparameterization f = K*a:
    log p(y, f) = log p(y|f) + log p(f)

    where log p(f) = -N/2 log(2π) - log|L| - 1/2 f^T K^{-1} f
    """
    f = self.K @ a  # Reparameterize

    # Log prior terms
    const = -self.N / 2 * jnp.log(2 * jnp.pi)
    logdet = jnp.sum(jnp.log(jnp.diag(self.L)))
    quad_term = 0.5 * jnp.sum(a * f)  # = 0.5 * f^T K^{-1} f
    log_prior = const - logdet - quad_term

    # Log likelihood
    log_lik = self.likelihood.log_lik(f)

    return log_prior + log_lik

def grad_a(self, a):
    """
    Gradient of log joint w.r.t. a:
    ∇_a log p(y, a) = K^T ∇_f log p(y, f) + ∇_a log p(f)
    """
    f = self.K @ a
    grad_prior = -f  # Derivative of prior term
    grad_lik = self.likelihood.grad(f) @ self.K  # Chain rule
    return grad_prior + grad_lik

def compute_f_MAP(self):
    """
    Find MAP estimate by optimization
    """
    result = minimize(lambda a: -self.log_joint_a(a),
                      jac=lambda a: -self.grad_a(a),
                      x0=jnp.zeros((self.N)))

    if not result.success:
        raise ValueError('Optimization failed')

    self.a = result.x
```

```
    f_MAP = self.K @ result.x
    return f_MAP
```

### 3.2 Posterior Covariance

The posterior covariance is computed using a numerically stable method:

python

```
def construct_laplace_approximation(self):
    """
    Constructs Laplace approximation: p(f|y) ≈ N(m, S)
    where S = (K^{-1} + Λ)^{-1}

    Uses Woodbury identity for numerical stability:
    S = K - K Λ^{1/2} (I + Λ^{1/2} K Λ^{1/2})^{-1} Λ^{1/2} K
    """
    # MAP estimate
    self.m = self.compute_f_MAP()

    # Negative Hessian of log likelihood
    Lambda = -self.likelihood.hessian(self.m)

    # Numerically stable computation
    Lsqrt = jnp.sqrt(Lambda)  # Element-wise square root
    B = jnp.identity(len(self.m)) + Lsqrt @ self.K @ Lsqrt
    chol_B = jnp.linalg.cholesky(B)
    e = jnp.linalg.solve(chol_B, Lsqrt @ self.K)

    # Posterior covariance
    self.S = self.K - e.T @ e
```

## 4. Prediction Methods

### 4.1 Predicting Latent Function Values

Predictive distribution for f* at new points:

python

```python
def predict_f(self, Xstar):
    """
    Posterior predictive distribution: p(f*|y, x*) ≈ N(μ*, Σ*)

    μ* = k*^T K^{-1} m
    Σ* = k** - k*^T K^{-1} (K - S) K^{-1} k*
    """
    # Cross-covariance between test and training points
    k_star_x = self.kernel.construct_kernel(Xstar, self.X, jitter=self.jitter)
    # Prior covariance at test points
    k_star_star = self.kernel.construct_kernel(Xstar, Xstar, jitter=self.jitter)

    # Solve linear systems instead of inverting
    h = jnp.linalg.solve(self.K, k_star_x.T)  # K^{-1} k*^T

    # Predictive mean
    mu = k_star_x @ jnp.linalg.solve(self.K, self.m)

    # Predictive covariance
    Sigma = k_star_star - h.T @ (self.K - self.S) @ h

    return mu, Sigma
```

## 4.2 Predicting Class Probabilities

Approximate integration for class probabilities:

```python
def predict_y(self, Xstar):
    """
    Predictive probability: p(y*=1|y, x*) ≈ Φ(μ*/√(c + σ*²))

    Uses probit approximation to sigmoid
    """
    mu, Sigma = self.predict_f(Xstar)

    # Probit approximation with variance adjustment
    # The factor 8/π comes from matching moments
    p = probit(mu / jnp.sqrt(8/jnp.pi + jnp.diag(Sigma)))

    return p
```

### 4.3 Posterior Sampling

Generate samples from the posterior:

```python
def posterior_samples(self, Xstar, num_samples):
    """
    Sample from p(f*|y, x*) ~ N(μ*, Σ*)
    """
    mu, Sigma = self.predict_f(Xstar)

    # Generate multivariate normal samples
    f_samples = generate_samples(mu.ravel(), Sigma, num_samples)

    return f_samples
```

## 5. Usage Example

Complete workflow for classification:

```python
# 1. Initialize kernel
kernel = StationaryIsotropicKernel(squared_exponential)

# 2. Create GPC model
gpc = GaussianProcessClassification(X, y, BernoulliLikelihood, kernel,
                                    kappa=3., lengthscale=1.)

# 3. Make predictions
# Latent function prediction
mu_f, Sigma_f = gpc.predict_f(Xtest)

# Class probability prediction
p_test = gpc.predict_y(Xtest)

# Binary predictions
y_pred = 1.0 * (p_test > 0.5)

# 4. Generate uncertainty samples
f_samples = gpc.posterior_samples(Xtest, num_samples=100)
p_samples = sigmoid(f_samples)
```

## 6. Mathematical Properties

### 6.1 Reparameterization Trick

The code uses f = K*a instead of working with f directly:

- Avoids inverting K during optimization
- Improves numerical stability
- Gradient becomes: ∇_a = K^T ∇_f

### 6.2 Woodbury Identity

For computing S = (K^{-1} + Λ)^{-1}:

- Direct inversion is $O(N^3)$ and numerically unstable
- Woodbury form avoids explicit K^{-1}
- Uses Cholesky decomposition of well-conditioned matrix B

### 6.3 Probit Approximation

Approximates sigmoid integral:

- $\sigma(f) \approx \Phi(f\sqrt{(\pi/8)})$
- Enables closed-form predictive probability
- Factor $\sqrt{(8/\pi + \sigma^2)}$ adjusts for uncertainty

## 7. Computational Considerations

- Training: $O(N^3)$ due to Cholesky decomposition
- MAP optimization: Iterative, typically $O(N^2)$ per iteration
- Prediction: $O(N^2)$ per test point
- Memory: $O(N^2)$ to store kernel matrix

The implementation prioritizes numerical stability over raw speed, using solver routines instead of explicit matrix inversions throughout.