# PML Sheet 6

Julian Zimmerlin 6003977, Leander Zimmermann 4165446

**Theory**

a) $\mathbb{E}_{p(y|\theta)}[y] = \int y \cdot p(y|\theta)\, dy = \int y \cdot \frac{1}{Z(\theta)} h(y) e^{y^T\theta}\, dy$

$Z(\theta) = \int h(y) e^{y^T\theta}\, dy$

$\Rightarrow \nabla_\theta \log Z(\theta) = \frac{1}{Z(\theta)} \cdot \nabla_\theta \left( \int h(y) e^{y^T\theta}\, dy \right)$

$\overset{(*)}{=} \frac{1}{Z(\theta)} \int h(y) e^{y^T\theta} \cdot y\, dy = \mathbb{E}_{p(y|\theta)}[y]$

where $(*)$ holds because $h(y)e^{y^T\theta}$ is continuously differentiable. Since this holds for all $\theta$, it also holds for $\theta = w^T x$.

b) $\log p(y|w,x) \overset{iid}{=} \log \prod_{n=1}^{N} p(y|w,x_n)$

$= \sum_{n=1}^{N} \log\left( \frac{1}{Z(\theta_n)} h(y) e^{y^T w^T x_n} \right)$

$= \sum_{n=1}^{N} -\log(Z(\theta_n)) + \log(h(y)) + y^T w^T x_n$

c) $\log p(y|w,X) = \sum_{n=1}^{N} -\log(Z(w^T x_n)) + \log(h(y)) + y w^T x_n$

(assuming $y$ is a scalar.)

$$\Rightarrow \frac{\partial}{\partial w} \log p(y|w,X) = \sum_{n=1}^{N} -\frac{1}{Z(w^T x_n)} \cdot \left( \frac{\partial Z}{\partial \theta}\Big|_{\theta = w^T x_n} \right) \cdot x_n^T + y \cdot x_n^T$$

$$\Rightarrow \nabla_w \log p(y|w,X) = \sum_{n=1}^{N} x_n \cdot \left( y - \frac{\left(\nabla_\theta Z(\theta)|_{\theta = w^T x_n}\right)}{Z(w^T x_n)} \right)$$

$$\overset{(a)}{=} \sum_{n=1}^{N} x_n \cdot \left( y - E_{p(y|\theta = w^T x_n)}[y] \right)$$

d) $\frac{\partial}{\partial w} \nabla_w \left( \log p(y|w,X) \right)$

$$= \frac{\partial}{\partial w} \left( \sum_{n=1}^{N} x_n \cdot \left( y - \frac{Z'(w^T x_n)}{Z(w^T x_n)} \right) \right)$$

$$= \sum_{n=1}^{N} -x_n \cdot \frac{Z''(w^T x_n) \cdot x_n^T \cdot Z(w^T x_n) - Z'(w^T x_n) \cdot x_n^T \cdot Z'(w^T x_n)}{\left(Z(w^T x_n)\right)^2}$$

$$= -\sum_{n=1}^{N} \frac{1}{\left(Z(w^T x_n)\right)^2} x_n \cdot \left( Z''(w^T x_n) Z(w^T x_n) - \left(Z'(w^T x_n)\right)^2 \right) \cdot x_n^T$$

$$= -\sum_{n=1}^{N} \frac{Z''(w^T x_n) Z(w^T x_n) - \left(Z'(w^T x_n)\right)^2}{\left(Z(w^T x_n)\right)^2} \cdot x_n x_n^T$$

$$\frac{\partial}{\partial w} \nabla_w \log p(w) = \frac{\partial}{\partial w} \left( \frac{1}{p(w)} \cdot \nabla_w p(w) \right)$$

$$= -\frac{1}{(p(w))^2} \cdot \nabla_w p(w) \cdot \frac{\partial p}{\partial w}(w) + \frac{1}{p(w)} \cdot \left( \frac{\partial}{\partial w} \nabla_w p(w) \right)$$

Assuming Gaussian prior: $p(w) = \mathcal{N}(w|0, \sigma^2 I)$:

$$\frac{\partial}{\partial w} \nabla_w \log p(w) = \frac{\partial}{\partial w} \nabla_w \left( -\frac{1}{2} w^T (\sigma^2 I)^{-1} w \right) = \frac{\partial}{\partial w} \left( -\sigma^{-2} \cdot w \right)$$

$$= -\sigma^{-2} \cdot I$$

$$\Rightarrow H = \frac{\partial}{\partial w} \nabla_w \left( \ln p(w|y,X) \right) = \frac{\partial}{\partial w} \nabla_w \left( \log p(y|w,X) + \log p(w) \right)$$

$$= -\sigma^{-2} \cdot I - \sum_{n=1}^{N} \frac{Z''(w^T x_n) Z(w^T x_n) - \left(Z'(w^T x_n)\right)^2}{\left(Z(w^T x_n)\right)^2} x_n x_n^T$$

# Exercise_06

June 3, 2022

## 1 Probabilistic Machine Learning

**Machine Learning in Science, University of Tübingen, Summer Semester 2022**

### 1.1 Exercise 06


### 1.2 Julian Zimmerlin 6009977

### 1.3 Leander Zimmermann 4165446


**hand in before 03.06.2022, 12:00 p.m. (noon)**

---

In the lecture you learned about Gaussian process regression, and how it is basically the infinite-dimension limit of classification using generalized linear models. In this programming exercise, you will explore GP classification, using publicly available implementations as well as implementing your own classifier.

### 1.4 Outline

1) Play around with a toy dataset and the scikit-learn black box GPs to get an intuition on how different kernels behave and ask high-level questions about Gaussian Processes, Kernels, etc. (easy part).

2) Implement a binary GP classification class yourself and apply it to the toy dataset and FashionMNIST dataset (harder part).

## 2 1) Binary Gaussian process classification using scikit-learn and a toy dataset

Recommended reference: https://scikit-learn.org/stable/modules/gaussian_process.html

Exercise adapted from https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpc.html#sphx-glr-download-auto-examples-gaussian-process-plot-gpc-py - License disclaimer

```
# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause
```

[1]:
```
import numpy as np

from matplotlib import pyplot as plt

from sklearn.metrics import accuracy_score, log_loss
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF, Matern, DotProduct,␣
 ↪RationalQuadratic, ExpSineSquared

np.random.seed(42)
```

### 2.1 Create a toy dataset $p(y, X)$

of $N$ samples

$$X \sim U(0,5) y(X) = \begin{cases} 0, & \text{if } X \leq 2.5 \\ 1, & \text{otherwise} \end{cases}$$

and split it into training and test splits. (Note: decide on your own how large $N$ and the split ratio is).

[2]:
```
# to create a toy dataset
n = 500
train_size = int(n * .8)

X = np.random.uniform(0,5, size=(n,1))
y = np.zeros(n)
y[X[:,0] > 2.5] = 1
X.shape, y.shape
```

[2]:
```
((500, 1), (500,))
```

[3]:
```
# to create a training and testing split
X_train = X[:train_size]
y_train = y[:train_size]
X_test = X[train_size:]
y_test = y[train_size:]
```

[4]:
```
# to verify the split
print("X_train shape: ", X_train.shape)
print("y_train shape: ", y_train.shape)
print("X_test shape: ", X_test.shape)
print("y_test shape: ", y_test.shape)
```

```
X_train shape:  (400, 1)
y_train shape:  (400,)
X_test shape:  (100, 1)
y_test shape:  (100,)
```

### 2.2 Specify kernels for the Gaussian process classification

- RBF
- Matern
- RationalQuadratic
- ExpSineSquared
- DotProduct

Tipp: check out the documentation in scikit-learn, e.g. for RBF https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.RBF.html

[5]:
```
# specify the kernels
RBF_kernel = 1.0 * RBF(1.0)
Matern_kernel = 1.0 * Matern(length_scale=1.0, nu=1.5)
RationalQuadratic_kernel = 1 * RationalQuadratic(length_scale=1.0, alpha=1.5)
ExpSineSquared_kernel = 1 * ExpSineSquared(length_scale=1, periodicity=1)
DotProduct_kernel = DotProduct()
```

### 2.3 Instantiate Gaussian process classifier (GPC) objects

Choose a kernel and create the Gaussian process classifier instances. To compare the initial kernel vs. the optimized kernel, create two instances: (1) `gp_fix` with `optimizer=None` and (2) `gp_opt` without specifying `optimizer`, leaving it at its default value (maximum log-marginal likelihood estimation). Call the `.fit(X, y)` method on both to fit the training data.

[6]:
```
# Specify Gaussian Processes with fixed and optimized hyperparameters
gp_fix = GaussianProcessClassifier(optimizer=None, kernel=RBF_kernel)
gp_fix.fit(X_train,y_train)

gp_opt = GaussianProcessClassifier(kernel=RBF_kernel)
gp_opt.fit(X_train,y_train)
```

```
/Users/leander/opt/anaconda3/lib/python3.8/site-
packages/sklearn/gaussian_process/_gpc.py:448: ConvergenceWarning: lbfgs failed
to converge (status=2):
ABNORMAL_TERMINATION_IN_LNSRCH.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  _check_optimize_result("lbfgs", opt_res)
/Users/leander/opt/anaconda3/lib/python3.8/site-
packages/sklearn/gaussian_process/kernels.py:411: ConvergenceWarning: The
optimal value found for dimension 0 of parameter k1__constant_value is close to
the specified upper bound 100000.0. Increasing the bound and calling fit again
may find a better value.
  warnings.warn("The optimal value found for "
```

[6]:
```
GaussianProcessClassifier(kernel=1**2 * RBF(length_scale=1))
```

[7]:
```
# to verify the kernels differ
gp_fix.kernel_, gp_opt.kernel_
```

[7]:
```
(1**2 * RBF(length_scale=1), 316**2 * RBF(length_scale=1.51))
```

### 2.4 Print the log marginal likelihood $\log p(\theta)$ for the kernel parameter to compare the initial and the optimized kernel parameter log probabilities

[8]:
```
print("Log Marginal Likelihood (initial): %.3f"
      % gp_fix.log_marginal_likelihood(gp_fix.kernel_.theta))
print("Log Marginal Likelihood (optimized): %.3f"
      % gp_opt.log_marginal_likelihood(gp_opt.kernel_.theta))
```

```
Log Marginal Likelihood (initial): -53.115
Log Marginal Likelihood (optimized): -6.603
```

### 2.5 Print the accuracy (using `accuracy_score`) between prediction and ground truth class to compare the initialized and the optimized GPC classification performance

[9]:
```
print("Accuracy: %.3f (initial) %.3f (optimized)"
      % (accuracy_score(y_test, gp_fix.predict(X_test)),
         accuracy_score(y_test, gp_opt.predict(X_test))))
```

```
Accuracy: 0.980 (initial) 1.000 (optimized)
```

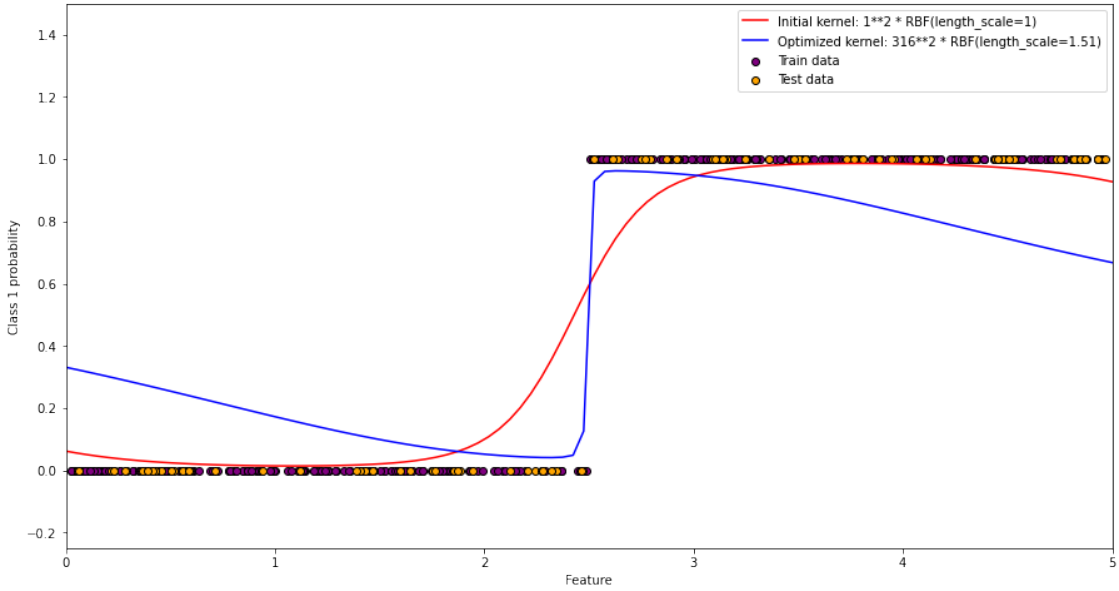
### 2.6 Print the log loss (using `log_loss`) between prediction and ground truth class to compare how the maximum log-marginal likelihood estimates compare to the arbitrary kernel parameters

[10]:
```
print("Log-loss: %.3f (initial) %.3f (optimized)"
      % (log_loss(y_test, gp_fix.predict_proba(X_test)),
         log_loss(y_test, gp_opt.predict_proba(X_test))))
```

```
Log-loss: 0.086 (initial) 0.198 (optimized)
```

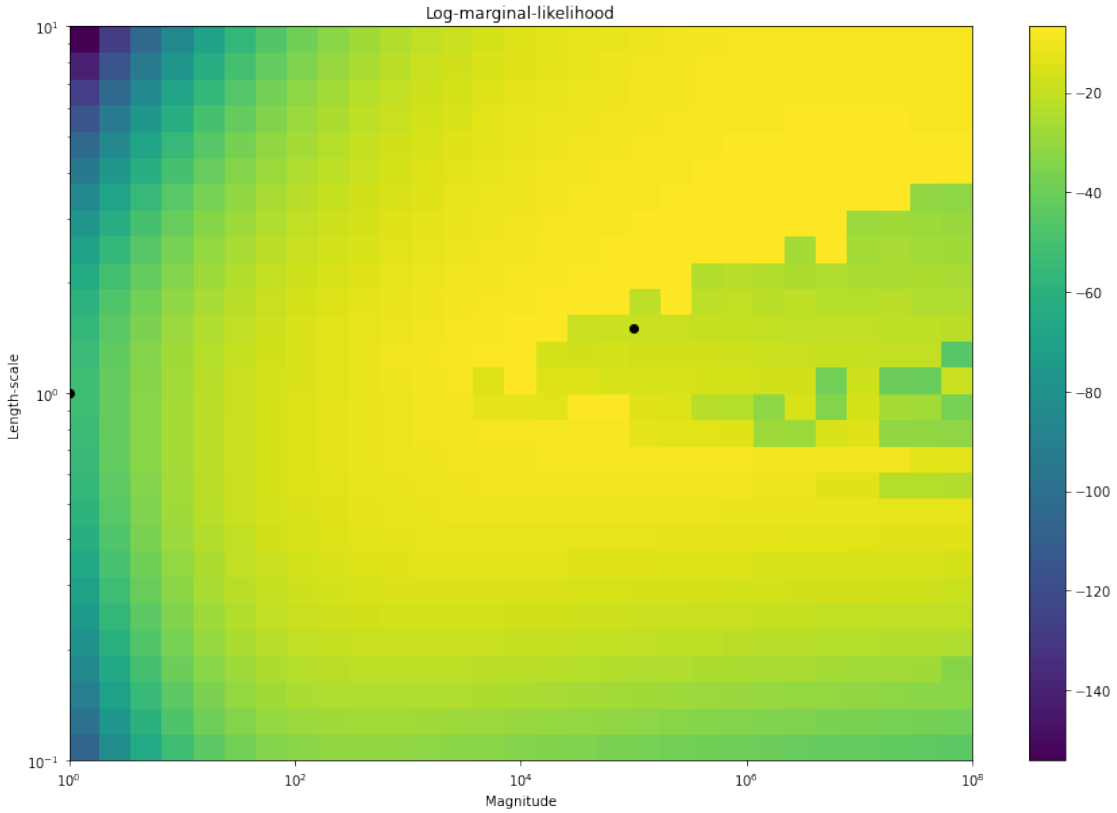
[11]:
```
# Plot posteriors
plt.figure(figsize=(15,8))
plt.scatter(X_train[:, 0], y_train, c='purple', label="Train data",
            edgecolors=(0, 0, 0))
plt.scatter(X_test[:, 0], y_test, c='orange', label="Test data",
            edgecolors=(0, 0, 0))
X_ = np.linspace(0, 5, 100)
plt.plot(X_, gp_fix.predict_proba(X_[:, np.newaxis])[:, 1], 'r',
         label="Initial kernel: %s" % gp_fix.kernel_)
plt.plot(X_, gp_opt.predict_proba(X_[:, np.newaxis])[:, 1], 'b',
         label="Optimized kernel: %s" % gp_opt.kernel_)
plt.xlabel("Feature")
```

```
plt.ylabel("Class 1 probability")
plt.xlim(0, 5)
plt.ylim(-0.25, 1.5)
plt.legend(loc="best")
plt.show()
```




[12]:
```
# Plot LML landscape
plt.figure(figsize=(15,10))
theta0 = np.logspace(0, 8, 30)
theta1 = np.logspace(-1, 1, 29)
Theta0, Theta1 = np.meshgrid(theta0, theta1)
LML = [[gp_opt.log_marginal_likelihood(np.log([Theta0[i, j], Theta1[i, j]]))
        for i in range(Theta0.shape[0])] for j in range(Theta0.shape[1])]
LML = np.array(LML).T
plt.plot(np.exp(gp_fix.kernel_.theta)[0], np.exp(gp_fix.kernel_.theta)[1],
         'ko', zorder=10)
plt.plot(np.exp(gp_opt.kernel_.theta)[0], np.exp(gp_opt.kernel_.theta)[1],
         'ko', zorder=10)
plt.pcolor(Theta0, Theta1, LML)
plt.xscale("log")
plt.yscale("log")
plt.colorbar()
plt.xlabel("Magnitude")
plt.ylabel("Length-scale")
plt.title("Log-marginal-likelihood")

plt.show()
```

```
<ipython-input-12-1181c4605a36>:13: MatplotlibDeprecationWarning: shading='flat'
when X and Y have the same dimensions as C is deprecated since 3.3.  Either
specify the corners of the quadrilaterals with X and Y, or pass shading='auto',
'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an
error two minor releases later.
  plt.pcolor(Theta0, Theta1, LML)
```




## 2.7 Questions:

1. What do the blue and the red line in the posterior plot represent?
2. What does the log-marginal-likelihood describe and what is it used for?
3. Using an initial RBF kernel `1**2 * RBF(length_scale=1)`, can you use the plot of the class 1 probability to reason about why the log loss with the initial RBF kernel is lower than with the optimized RBF kernel?
4. Check out all 5 different kernels. Which one is the best fit? Is any Kernel strictly better than another?
5. Implement a different function and test it with all 5 Kernels. Is every Kernel able to fit the problem? The function can be simple, e.g. 1 if $x < 1$ or $x > 4$, 0 else.

Note: text of single sentences to answer the questions that don't require code at this point are sufficient.

### 2.8 Answers:

1. Both show the output probability (for class 1) of the corresponding estimator, over the given sample X. The blue line is using the optimized Kernel, which results in a much sharper deision boundary
2. It is used to judge how well we fit the data with a given kernel and parameters. The higher the marginal likelihood, the better the fit of the data.
3. While in the middle, the optimized kernel outputs probabilities closer ot 0 and 1, towards X=0 and X=5, it is the other way around, s.t. the optimized kernel produces probabilities close to .5 overall, resulting in a higher loss inspite of the perfect acuracy.
4. Most Kernels produce a really good accuracy. In terms of log loss the optimized DotProduct Kernel, followed by the unoptimized MaternKernel perform the best. The unoptimized ExpSineSquared Kernel is not usable for this problem

## 3 2) Implementing a binary Gaussian process classification object

Instructions: - complete the methods `fit`, `predict_all` of the class `GPClassifier`. This is supposed to give some structure to the programming to focus on the hard parts. Feel free to implement your own way from scratch. We suggest you document *why* you implement it the way you do.

```
from scipy.stats import norm
from scipy.integrate import quad
from scipy.special import expit as sigmoid
from scipy.spatial.distance import cdist

# based on the implementation given in the book by Rasmussen and Williams
class GPClassifier:

    def __init__(self, X, y, kernel):
        self.X = X
        self.y = y
        self.kernel = kernel
        self.K = self.kernel(self.X, self.X)
        assert len(self.X) == len(self.y)
        self.size = len(self.X)

        self.f = None
        self.W = None
        self.L = None

    def fit(self):
        """Single class, mode-finding algorithm for binary GPC.

        Returns:
             float: Log-marginal likelihood after convergence,
                 here |obj - obj_old| < 1e-5.

        Note:
```

```
        Requires:
            self.K
        Updates:
            self.W
            self.L
            self.f

    Reference: Algorithm 3.1, Rasmussen and Williams
    """
    ...


def predict_all(self, X_pred):
    """Predictions for binary GPC.

    Returns:
        array: predictive class probability for class 1.
        array: class labels y_pred.

    Note:
        Requires:
            self.X
            self.y
            self.L
            self.W
            self.kernel
            self.f

    Reference: Algorithm 3.2, Rasmussen and Williams
    """
    ...
```

- only consider binary classification
- use the cholesky decomposition to solve the linear system because that is numerically stable
- use the kernel parameters suggested in the toy example instead of implementing an optimizer
- Hints:
    1. Use the knowledge from the lecture on GPs and Gaussian_Process_Regression_2022.ipynb. You can base your implementation on the classification algorithm (3.1) from Rasmussen and Williams in http://www.gaussianprocess.org/gpml/chapters/RW3.pdf.
    2. Implement the likelihood according to using [0,1] labels.
    3. You can compute a definite integral using `scipy.integrate.quad`.

[13]:
```
# implement your GPClassifier here
from scipy.stats import norm
from scipy.integrate import quad
```

```python
from scipy.special import expit as sigmoid
from scipy.spatial.distance import cdist
from numpy.linalg import cholesky, solve


# based on the implementation given in the book by Rasmussen and Williams
class GPClassifier:

    def __init__(self, X, y, kernel):
        self.X = X
        self.y = y
        self.kernel = kernel
        self.K = self.kernel(self.X, self.X)
        assert len(self.X) == len(self.y)
        self.size = len(self.X)

        self.f = None
        self.W = None
        self.L = None

    def fit(self):
        """Single class, mode-finding algorithm for binary GPC.

        Returns:
             float: Log-marginal likelihood after convergence,
                 here |obj - obj_old| < 1e-5.

        Note:
            Requires:
                self.K
            Updates:
                self.W
                self.L
                self.f

        Reference: Algorithm 3.1, Rasmussen and Williams
        """
        f = np.zeros(self.size)
        log_marginal_likelihood = -np.inf
        while True:
            # Line 4
            pi = sigmoid(f)
            W = np.zeros_like(self.K)

            np.fill_diagonal(W, (pi * (1-pi)))
            W_sqrt = np.sqrt(W)
```

```
            # Line 5
            B = np.identity(W.shape[0]) + W_sqrt @ self.K @ W_sqrt

            L = np.linalg.cholesky(B)

            # Line 6
            b = W @ f + (self.y - pi)

            # Line 7
            a = b - W_sqrt @ solve(L.T, solve(L, W_sqrt @ self.K @ b))

            # Line 8
            f = self.K @ a

            # Line 10:
            lml = (-0.5 * a.T @ f) - np.log(1 + np.exp(-((self.y*2)-1) * f)).
↪sum() - np.log(L.diagonal()).sum()

            # Convergence check using log marginal LL
            if np.abs(lml - log_marginal_likelihood) < 1e-5:
                break
            log_marginal_likelihood = lml

        self.f = f
        self.W = W
        self.L = L
        return log_marginal_likelihood

    def predict_all(self, X_pred):
        """Predictions for binary GPC.

        Returns:
            array: predictive class probability for class 1.
            array: class labels y_pred.

        Note:
            Requires:
                self.X
                self.y
                self.L
                self.W
                self.kernel
                self.f

        Reference: Algorithm 3.2, Rasmussen and Williams
        """
        f_star = self.kernel(self.X, X_pred).T @ (self.y - sigmoid(self.f))
```

```
        v = solve(self.L, np.sqrt(self.W) @ self.kernel(self.X, X_pred))
        V_fstar = self.kernel(X_pred,X_pred) - v.T @ v

        pi_s = []
        for i in range(len(f_star)):
            mean = f_star[i]
            var = V_fstar.diagonal()[i]
            pi_s.append(quad(lambda z: sigmoid(z)*norm.pdf(z, loc=mean,␣
↪scale=var), -np.inf, np.inf)[0])
        scores = np.array(pi_s)
        preds = (np.array(pi_s) >= 0.5).astype(int)

        return (scores, preds)
```

### 3.1 Use your own GPC implementation to classify the toy data from exercise 1)

Instructions: - implement the rbf kernel (and e.g. linear) to be used with `GPClassifier`. E.g.: “‘ python # kernels: def rbf(x1,x2, sigma, ell): … def linear(x_1, x_2, sigma): …

```
# set the kernel parameters from the toy example
rbf_kernel = lambda x1, x2: rbf(x1, x2, 91.7, 1.12)
linear_kernel = lambda x1, x2: linear(x1, x2, ?)
```

- compare your results to the scikit-learn off-the-shelf GP. Hint: they should be similar ;)

[14]:
```
# to create a toy dataset
n = 100
train_size = 50
X = np.random.uniform(0, 5, n)
y = (X > 2.5).astype(int)  # using bool to int casting
# to create a training and testing split
X_train = X[0:train_size].reshape(-1,1)
y_train = y[0:train_size]
X_test = X[train_size:n].reshape(-1,1)
y_test = y[train_size:n]
# to verify the split
print("X_train shape: ", X_train.shape)
print("y_train shape: ", y_train.shape)
print("X_test shape: ", X_test.shape)
print("y_test shape: ", y_test.shape)
```

```
X_train shape:  (50, 1)
y_train shape:  (50,)
X_test shape:  (50, 1)
y_test shape:  (50,)
```

[15]:
```
# implement your kernels here and classify the toy data
# kernels:
def rbf(x1,x2, sigma, ell):
    # here we use the identity ||x-y||^2 = ||x||^2 + ||y||^2 - 2 * x^T * y
    x1_norm = np.sum(x1**2, axis=-1)
    x2_norm = np.sum(x2**2, axis=-1)
    K = sigma**2 * np.exp(-1/(2*ell**2) * (x1_norm[:,None] + x2_norm[None,:] -␣
 ↪2 * np.dot(x1, x2.T)))
    return K

def linear(x_1, x_2, sigma):
    return sigma**2 * x_1 @ x_2.T


## RBF

# OUR CODE
rbf_kernel = lambda x1, x2: rbf(x1, x2, 91.7, 1.12)
gpc = GPClassifier(X_train, y_train, rbf_kernel)
gpc.fit()
scores, preds = gpc.predict_all(X_test)
print(scores)

# STANDARD SKLEARN
gp_opt = GaussianProcessClassifier(kernel=RBF_kernel)
gp_opt.fit(X_train,y_train)

print("RBF KERNEL -- Accuracy: %.3f (OURS) %.3f (SKLEARN)"
      % (accuracy_score(y_test, preds),
         accuracy_score(y_test, gp_opt.predict(X_test))))
```

```
[0.53311262 0.50244811 0.52697196 0.52400962 0.5023418  0.48301204
 0.4982252  0.49905416 0.50350383 0.51119994 0.50492751 0.49725559
 0.49820345 0.49915331 0.49115444 0.51634483 0.48534079 0.47334459
 0.50224739 0.49138245 0.53022292 0.50304029 0.48449731 0.51096856
 0.50239977 0.50204997 0.4981011  0.5021652  0.49513889 0.49625921
 0.46717544 0.50181248 0.49617974 0.49300881 0.50963659 0.49671904
 0.49870565 0.49827172 0.49827917 0.49805275 0.4981805  0.50887606
 0.49771285 0.49162631 0.50227192 0.46958594 0.50676455 0.49782982
 0.4975742  0.49896733]
RBF KERNEL -- Accuracy: 1.000 (OURS) 1.000 (SKLEARN)
```

[16]:
```
## LINEAR:

# OUR CODE
linear_kernel = lambda x1, x2: linear(x1, x2, 0.006)
gpc = GPClassifier(X_train, y_train, linear_kernel)
```

```
gpc.fit()
scores, preds = gpc.predict_all(X_test)

# STANDARD SKLEARN
gp_opt = GaussianProcessClassifier(kernel=DotProduct_kernel)
gp_opt.fit(X_train,y_train)

print("LINEAR KERNEL -- Accuracy: %.3f (OURS) %.3f (SKLEARN)"
      % (accuracy_score(y_test, preds),
         accuracy_score(y_test, gp_opt.predict(X_test))))
```

```
LINEAR KERNEL -- Accuracy: 0.660 (OURS) 0.960 (SKLEARN)
```

We can see that we get similar accuracies to the Sklearn implentation with both RBF (1.0/1.0) and linear kernels (0.94 / 0.98).

### 3.2 Use your own GPC implementation to separate images of Tshirts from images of Trousers using Fashion MNIST

Instructions: - we use class 0 for t-shirts/tops and class 1 for trousers as the two classes for the binary classification. - the nasty data wrangling stuff has already been done for you by Pytorch and us. - decide on a considerate number of samples to train the GPC. - how accurate can you separate images of Tshirts from images of Trousers using your own `GPClassifier`?

[19]:
```
import torch, torchvision
import torchvision.transforms as transforms
torch.manual_seed(42)
```

[19]:
```
<torch._C.Generator at 0x7f7de97d5790>
```

[20]:
```
# data wrangling: get the first two classes for FMNIST
FMNIST_transform = transforms.Compose([
    transforms.ToTensor(),
])


FMNIST_train = torchvision.datasets.FashionMNIST(
        '~/data/fmnist',
        train=True,
        download=True,
        transform=FMNIST_transform)


FMNIST_test = torchvision.datasets.FashionMNIST(
        '~/data/fmnist',
        train=False,
        download=True,
        transform=FMNIST_transform)
```

```
def get_subset_FMNIST(classes=[0,1], dataset=FMNIST_train):

    dataset.targets = dataset.targets.clone().detach()
    idx = torch.zeros(dataset.targets.size()).byte()
    for c in classes:
        idx += (dataset.targets==c).byte()
    dataset.targets= dataset.targets[idx]
    dataset.data = dataset.data[idx.numpy().astype(np.bool)]

    #remap all classes to list of ints starting at 1
    map_dict = dict()
    for i, c in enumerate(classes):
        map_dict[c] = i

    for i, t in enumerate(dataset.targets):
        dataset.targets[i] = map_dict[t.item()]

    return(dataset.data, dataset.targets)

classes = [0,1]
FMNIST_train.data, FMNIST_train.targets = get_subset_FMNIST(classes=classes,␣
 ↪dataset=FMNIST_train)
FMNIST_test.data, FMNIST_test.targets = get_subset_FMNIST(classes=classes,␣
 ↪dataset=FMNIST_test)
print(len(FMNIST_train.targets))
print(FMNIST_train.targets[:10])
print(len(FMNIST_test.targets))
print(FMNIST_test.targets[:10])
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to /Users/leander/data/fmnist/FashionMNIST/raw/train-
images-idx3-ubyte.gz
```

```
  0%|          | 0/26421880 [00:00<?, ?it/s]
```

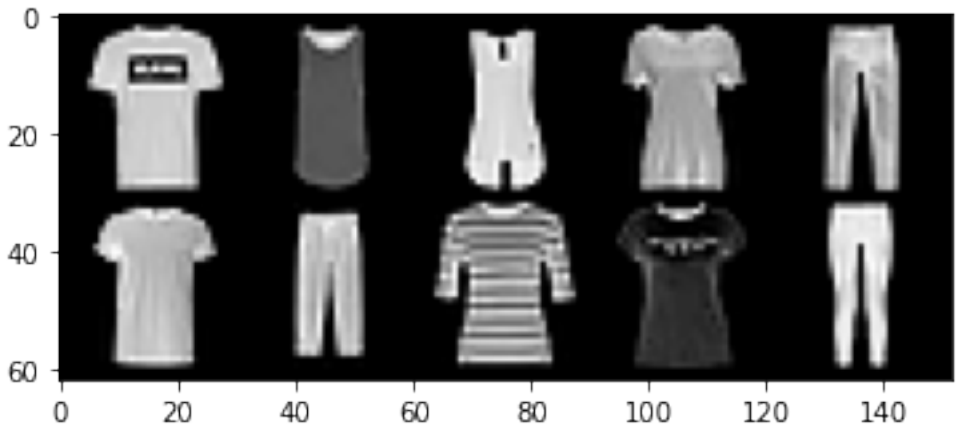
```
Extracting /Users/leander/data/fmnist/FashionMNIST/raw/train-images-
idx3-ubyte.gz to /Users/leander/data/fmnist/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to /Users/leander/data/fmnist/FashionMNIST/raw/train-
labels-idx1-ubyte.gz
```

```
  0%|          | 0/29515 [00:00<?, ?it/s]
```

```
Extracting /Users/leander/data/fmnist/FashionMNIST/raw/train-labels-
```

```
idx1-ubyte.gz to /Users/leander/data/fmnist/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
/Users/leander/data/fmnist/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/4422102 [00:00<?, ?it/s]

Extracting /Users/leander/data/fmnist/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
to /Users/leander/data/fmnist/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
/Users/leander/data/fmnist/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/5148 [00:00<?, ?it/s]

Extracting /Users/leander/data/fmnist/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
to /Users/leander/data/fmnist/FashionMNIST/raw

12000
tensor([0, 0, 0, 0, 1, 0, 1, 0, 0, 1])
2000
tensor([1, 1, 1, 1, 0, 1, 0, 0, 1, 1])

<ipython-input-20-051cf280937c>:26: UserWarning: indexing with dtype torch.uint8
is now deprecated, please use a dtype torch.bool instead. (Triggered internally
at  /Users/distiller/project/pytorch/aten/src/ATen/native/IndexingUtils.h:28.)
  dataset.targets= dataset.targets[idx]
<ipython-input-20-051cf280937c>:27: DeprecationWarning: `np.bool` is a
deprecated alias for the builtin `bool`. To silence this warning, use `bool` by
itself. Doing this will not modify any behavior and is safe. If you specifically
wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  dataset.data = dataset.data[idx.numpy().astype(np.bool)]
```

[21]:
```
# have a look at the images corresponding to FMNIST_train.targets[:10]
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    #plt.imshow(npimg)
    plt.show()

images = FMNIST_train.data[:10].view(10, 1, 28, 28)
imshow(torchvision.utils.make_grid(images, nrow=5))
```




[50]:
```
# further data wrangling

n_train_samples = 20
n_test_samples = 20

X_train = FMNIST_train.data[:n_train_samples].view(-1, 28*28)
y_train = FMNIST_train.targets[:n_train_samples].view(-1)

X_test = FMNIST_test.data[:n_test_samples].view(-1, 28*28)
y_test = FMNIST_test.targets[:n_test_samples].view(-1)

#normalize the data between [-1,1]
X_train = (X_train.float() - 128) / 128
X_test = (X_test.float() - 128) / 128

X_train, X_test = X_train.numpy(), X_test.numpy()
y_train, y_test = y_train.numpy(), y_test.numpy()
```

[51]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(20, 784)
(20, 784)
(20,)
(20,)
```

[52]:
```
# classify the data here and report your accuracy

rbf_kernel = lambda x1, x2: rbf(x1, x2, 91.7, 1.12)
gpc = GPClassifier(X_train, y_train, rbf_kernel)
```

```
print('Fitting...')
gpc.fit()
print('Fitted. Predicting...')
scores, preds = gpc.predict_all(X_test)
print('Predicted')
print("Accuracy: %.3f (OURS)"
      % (accuracy_score(y_test, preds)))
```

```
Fitting…
Fitted. Predicting…
Predicted
Accuracy: 0.800 (OURS)
```

Unfortunately, fitting and prediiting the data would have taken extremely long with the full dataset, which is why we tookonly 20 train and 20 test samples. Therefore the predicitor can not be very good

[ ]: