main

May 14, 2020

```
In [61]: import numpy as np
         import importlib
         import pandas as pd
         import sklearn as skl
         import util
         from scipy.special import expit
         from scipy.optimize import root
         from sklearn import gaussian_process as gp
         from sklearn import linear_model as lm
         from sklearn.decomposition import PCA
         from sklearn.svm import SVC
         from sklearn.neural_network import MLPClassifier
```

1 Boosting Classification Performance with Gaussian Processes

We use the scikit-learn package to implement Gaussian Classification on a selected COVID-19 data set, and we compare performance against benchmark models.

Preliminary Analysis

Gaussian Processes

First, we implement Gaussian Process Classification on the CT scans of the lungs of hospital patients that fall into three categories:

- Normal Patients
- Patients with Viral Pneumonia
- Patients infected with COVID-19

```
In [14]: num_images = int(10e6)
         images, class_labels = util.load_covid_images(num_images)
         NUM_IMAGES = images.shape[0]
```

We use our custom util module to load all of the scans, of which there are at most *num_images* in each category. We assign a label of

- 0 for the patients infected with COVID-19
- 1 for the normal patients
- 2 for the patients with viral pneumonia

and collect these in the variable *class_labels*.

```
In [15]: d = 1000
         pca = PCA(n_components=d)
         reduced_images = pca.fit_transform(images)
```

Each image is a vector consisting of 1024 x 1024 x 3 Red/Green/Blue Values. It is computationally intractable to perform GP classification using the image vectors because their dimension is too large. Therefore, we choose to project these vectors into a lower dimensional space using principal component analysis, where we choose d = 1000 principal components. The variable *reduced_images* contains the coordinates of each vector in this lower dimensional space.

```
In [16]: P = np.random.permutation(np.identity(NUM_IMAGES))
         reduced_images = P @ reduced_images
         class_labels = P @ class_labels
         n_train = int(.8*NUM_IMAGES)
         X_train = reduced_images[:n_train]
         Y_train = class_labels[:n_train]
         X_test = reduced_images[n_train:]
         Y_test = class_labels[n_train:]
```

In order to ensure a different test set for each replication of the experiment, we randomly permute the reduced image set. We use an 80 / 20 split for the training and test data.

```
In [17]: gp_classifier = gp.GaussianProcessClassifier()
         gp_classifier.fit(X_train, Y_train)

Out[17]: GaussianProcessClassifier(copy_X_train=True, kernel=None, max_iter_predict=100,
                                   multi_class='one_vs_rest', n_jobs=None,
                                   n_restarts_optimizer=0, optimizer='fmin_l_bfgs_b',
                                   random_state=None, warm_start=False)

In [18]: print("GP Accuracy:", gp_classifier.score(X_test, Y_test))

GP Accuracy: 0.4974182444061962
```

Voila! We've trained the GP classifier on the reduced image set and associated class labels. We have specified a zero mean prior and a squared exponential covariance function. The result is a correct prediction on approximately 50% of the test set. Recall that the expected accuracy of guessing randomly would be 33%, since there are three classes. We now train a support vector machine, a feedforward neural network, and a logistic model on the same image set and compare results.

Support Vector Machine

```
In [19]: svc = SVC()
         svc.fit(X_train, Y_train)

Out[19]: SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
             decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
             max_iter=-1, probability=False, random_state=None, shrinking=True,
             tol=0.001, verbose=False)
```

```
In [20]: print("SVM Accuracy:", svc.score(X_test, Y_test))
```

```
SVM Accuracy: 0.9466437177280551
```

Neural Network

```
In [21]: mlp_classifier = MLPClassifier()
         mlp_classifier.fit(X_train, Y_train)
```

```
Out[21]: MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
                       beta_2=0.999, early_stopping=False, epsilon=1e-08,
                       hidden_layer_sizes=(100,), learning_rate='constant',
                       learning_rate_init=0.001, max_fun=15000, max_iter=200,
                       momentum=0.9, n_iter_no_change=10, nesterovs_momentum=True,
                       power_t=0.5, random_state=None, shuffle=True, solver='adam',
                       tol=0.0001, validation_fraction=0.1, verbose=False,
                       warm_start=False)
```

```
In [22]: print("Neural Net Accuracy:", mlp_classifier.score(X_test, Y_test))
```

```
Neural Net Accuracy: 0.882960413080895
```

Logistic Model

```
In [24]: logistic_regressor = lm.LogisticRegression()
         logistic_regressor.fit(X_train, Y_train)
```

```
/Users/Zhonghou/anaconda3/lib/python3.6/site-packages/sklearn/linear_model/_logistic.py:940: C
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

```
Out[24]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                            intercept_scaling=1, l1_ratio=None, max_iter=100,
                            multi_class='auto', n_jobs=None, penalty='l2',
                            random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                            warm_start=False)
```

```
In [25]: print("Logistic Regression Accuracy:", logistic_regressor.score(X_test, Y_test))
```

```
Logistic Regression Accuracy: 0.9053356282271945
```

These results indicate that the performance of GP classification is suboptimal and is beaten by every other model. How can we improve performance? We identify two problems and suggest potential solutions.

1. The implementation of GP Classification depends on a Laplace approximation to the posterior distribution of the process f given training set *X_train* and labels *Y_train*, which may not be satisfactory if the posterior is multimodal.

2. Our choice of the squared exponential kernel for the GP classifier may not be optimal.

For our purposes, we find it sufficient to devote our focus to the second problem, so we explore several other kernels and select the one which maximizes the marginal likelihood of (*X_train, Y_train*) as well as the one that minimizes the PAC generalization error bound.

Marginal Likelihoood

```
In [83]: import sklearn.gaussian_process.kernels as kernels
         model_kernels = [kernels.ConstantKernel(), kernels.Matern(),
                          kernels.RationalQuadratic(), kernels.WhiteKernel()]
         num_kernels = len(model_kernels)
         test_accuracies = np.empty(num_kernels)
         log_marginal_likelihoods = np.empty(num_kernels)
```

We start by loading four kernels from sklearn's collection and we initialize two arrays to store the test errors for each model and the marginal likelihoods of the training set under each model.

```
In [85]: for kernel_num in range(num_kernels):
             kernel = model_kernels[kernel_num]
             gp_classifier = gp.GaussianProcessClassifier(kernel=kernel)
             gp_classifier.fit(X_train, Y_train)
             log_marginal_likelihoods[kernel_num] = gp_classifier.log_marginal_likelihood()
             test_accuracies[kernel_num] = gp_classifier.score(X_test, Y_test)

         best_classifier_num = np.argmax(log_marginal_likelihoods)
```

We iterate over the kernels and store the test errors and marginal likelihoods, and we find the kernel and corresponding classifier that maximizes the marginal likelihood.

```
In [54]: print("Test Accuracy:", test_errors[best_classifier_num])

Test Accuracy: 0.9397590361445783
```

As we can see this GP Classifier's performance far surpasses that of its predecessor and does about as well as the Support Vector Machine. We now need only compare our results against a different scheme, where we minimize the PAC generalization error bound instead.

PAC Generalization Error

```
In [112]: def compute_KL(kernel, X, Y):
              K = kernel(X, X)
              n = K.shape[0]
```

```
        f0 = np.zeros(n)
        def system(f):
            pi = expit(f)
            return f - K @ (Y - pi)
        f_hat = root(system, f0, method='broyden1').x
        pi = expit(f_hat)
        W = np.identity(n)
        for i in range(n):
            W[i][i] = pi[i] * (1 - pi[i])
        eps = 10e-3
        K_inv = la.inv(K + eps*np.identity(n))
        A = K_inv + W
        kl = 1/2*np.log(abs(la.det(K))+eps)
        kl += 1/2*np.log(abs(la.det(A))+eps)
        kl += 1/2*np.trace(la.inv(A+eps*np.identity(n))@(K_inv - A))
        kl += np.reshape(f_hat, (1, n)) @ K_inv @ f_hat
        return kl

    kl_divergences = np.zeros(num_kernels)
    for kernel_num in range(num_kernels):
        for c in range(3):
            Y = (Y_train == c).astype(int)
            kl_divergences[kernel_num] += 1/3*compute_KL(kernel, X_train, Y)

    kl_best_classifier_num = np.argmin(kl_divergences)
```

In accordance with Seeger 2003, the PAC generalization error is bounded by an increasing function of the KL divergence between the prior and posterior distributions over the possible function values at the test and training points. Therefore, if we wish to minimize this error bound, we can minimize the KL divergence. In fact, the KL divergence can be computed by the function above, as noted in Rasmussen and Williams. We compute the KL divergence under each model, and choose the kernel for which it is minimized.

```
In [100]: print("Test Accuracy:", test_accuracies[kl_best_classifier_num])

Test Accuracy: 0.44922547332185886
```

The result is a poor test accuracy, as Seeger predicted. In the future, we will heed his advice and use the marginal likelihood, cross validation, or bayesian model selection to distinguish between models, resorting to the PAC bounds only if these other tools are unavailable.