Documentation: Adaptive Transfer Learning with Gaussian Processes (AT-GP)

Authors of the paper: Bin Cao, Jialin Pan, Yu Zhang, Dit-Yan Yeung, Qiang Yang
Hong Kong University of Science and Technology

Implementation by: Harshdeep J.
SE21UARI044

May 31, 2025

Abstract

Many Transfer Learning methods assume that the source and the target tasks are related, even though there are many tasks which might not be related in reality. In such a case, the knowledge extracted from the source task may not help, or might even hurt the performance of the target task. Thus, to avoid this negative transfer, the authors of the paper propose an Adaptive Transfer Learning approach based on Gaussian Processes which can automatically estimate the similarity between the source task and the target task by proposing a semi-parametric transfer kernel for transfer learning.

The code for this implementation is available on GitHub: ATGP-implementation.

Contents

1 Prerequisite

1.1 Gaussian Processes

Gaussian Processes are non-parametric Bayesian approach used for regression and probabilistic modeling. This allows us to define a distribution over functions, rather than just parameters, so what GP gives is a distribution of all possible functions that could probably explain the data while giving a measure of uncertainty for its prediction.

1.1.1 Math Behind It

For simplicity, we can assume that the mean function $m(x) = 0$ and the covariance function (kernel) is $k(x, x')$.

A Gaussian process is defined by its mean function and its covariance kernel.

$$f(x) \sim \mathcal{GP}(m(x), k(x, x'))$$

The kernel function measures the similarity between any two input points x and x'. The choice of kernel is crucial as it encodes our assumption about the smoothness, periodicity, or other properties of the function we are trying to model.

A very common kernel is Squared Exponential kernel:

$$k(x_i, x_j) = \sigma_f^2 \exp\left(-\frac{||x_i - x_j||^2}{2l^2}\right)$$

Where:

1. x_i and x_j are two input points.
2. $||x_i - x_j||^2$ is the squared Euclidean distance between x_i and x_j.
3. σ_f^2 (signal variance) controls the overall vertical scale of the function.
4. l (length scale) controls how quickly the correlation decays with distance. A large l leads to a smoother function.

Now, given a set of training input points $X = [x_1, x_2, ..., x_N]^T$, the corresponding function values $f(X) = [f(x_1), f(x_2), ..., f(x_N)]^T$ are jointly Gaussian distributed:

$$f(X) \sim N(\mathbf{0}, K(X, X))$$

Where:

1. $\mathbf{0}$ is a vector of zeros.
2. $K(X, X)$ is a $N \times N$ covariance matrix, where each element $K_{ij} = k(x_i, x_j)$. This matrix captures the prior relationships between all pairs of training data points based on the chosen kernel.

In real-world data, observations are usually noise. We model this by addidng Gaussian noise to the true function values:

$$y_i = f(x_i) + \epsilon$$

Where $\epsilon \sim N(0, \sigma_n^2)$ is independent and identically distributed Gaussian noise with σ_n^2 (noise variance).

So, our observed training outputs y_{train} are also jointly Gaussian distributed:

$$y_{train} \sim N(\mathbf{0}, K(X_{train}, X_{train}) + \sigma_n^2 \boldsymbol{I})$$

where $\boldsymbol{I}$ is the identity matrix.

Now, when we get a new test input point x_*, we want to predict its corresponding output y_*. We form a joint distribution of the training outputs y_{train} and the test output $f(x_*)$

$$\begin{bmatrix} y_{\text{train}} \\ f(x_*) \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X_{\text{train}}, X_{\text{train}}) + \sigma_n^2 I & K(X_{\text{train}}, x_*) \\ K(x_*, X_{\text{train}}) & k(x_*, x_*) \end{bmatrix}\right)$$

Here, $K(X_{\text{train}}, X_{\text{train}})+\sigma_n^2 I$ is the covariance of the noisy training outputs y_{train}. $K(X_{\text{train}}, x_*)$ is the covariance between training points and the test point function value $f(x_*)$, $K(x_*, X_{train})$ is its transpose, and $k(x_*, x_*)$ is the prior variance of the test point function value $f(x_*)$.

While prediction we are interested in the conditional distribution $f(x_*)$, given the observed training data y_{train}. For multivariate Gaussian distribution, the conditional distribution of one block of variables given another is also a Gaussian distribution.

The posterior predictive distribution for $f(x_*)$ is:

$$f(x_*)|X_{train}, y_{train}, x_* \sim N(\mu_{post}, \Sigma_{post})$$

Where:

1. $\mu_{post} = K(x_*, X_{\text{train}})(K(X_{\text{train}}, X_{\text{train}}) + \sigma_n^2 I)^{-1} y_{\text{train}}$
2. $\Sigma_{post} = k(x_*, x_*) - K(x_*, X_{\text{train}})(K(X_{\text{train}}, X_{\text{train}}) + \sigma_n^2 I)^{-1} K(X_{\text{train}}, x_*)$

Note: $K_{*,\text{train}}$ is shorthand for $K(x_*, X_{\text{train}})$, $K_{\text{train,train}}$ for $K(X_{\text{train}}, X_{\text{train}})$, and $K_{*,*}$ for $k(x_*, x_*)$.

And while predicting $y_* = f(x_*) + \epsilon_*$, where $\epsilon_* \sim N(0, \sigma_n^2)$, the predictive variance for y_* becomes:

$$\text{Var}(y_*|X_{train}, y_{train}, x_*) = \Sigma_{post} + \sigma_n^2$$

2 Proposed Work

The following modifications are done in standard Gaussian distribution in this work:

1. Instead of modeling the joint distribution of training outputs and test function values and then conditioning on the training outputs y_{train} to obtain the posterior distribution of the test prediction $f(x_*)$, the authors optimize the conditional distribution:

$$p(y^{(T)}|y^{(S)}, X^{(T)}, X^{(S)})$$

2. Instead of using the fixed kernel $K_{ij} = k(x_i, x_j)$, a semi-parametric kernel is introduces:

$$K_{nm} \sim k(x_n, x_m)(2e^{-\zeta(x_n, x_m)\rho} - 1)$$

where $\zeta(x_n, x_m) = 0$ if the points are from the same task, and 1 if not. This adds a task-similarity factor that modulates how much the GP kernel links points from different tasks based on an inferred dissimilarity parameter ρ. Here ρ is modeled as a random variable,

$$\rho \sim \Gamma(b, \mu)$$

where $\Gamma(b, \mu)$ is a Gamma distribution with shape b and mean parameter μ. This gives a closed-form kernel expression:

$$K_{nm} = \begin{cases} k(x_n, x_m) \cdot \left(2\left(\frac{1}{1+\mu}\right)^b - 1\right), & \text{if tasks differ} \\ k(x_n, x_m), & \text{if same task} \end{cases}$$

3. The final predictive mean is:

$$m(x) = \sum_{x_j \in X^{(T)}} \alpha_j k(x, x_j) + \sum_{x_i \in X^{(S)}} \lambda \alpha_i k(x, x_i)$$

Where $\lambda = 2(1/(1+\mu))^b - 1$, quantifies:

- No Transfer when $\lambda \to 0$,
- Partial Transfer when $0 < |\lambda| < 1$,
- Full Transfer when $\lambda = 1$.

Hence, all the hyperparameters, including μ, b and kernel parameters are learnt by maximizing the conditional likelihood of the target outputs given the source data:

$$\log p(y^{(T)} | y^{(s)}, X^{(T)}, X^{(S)})$$

instead of maximizing the marginal likelihood over all observed data.

3 Implementation

3.1 Standard Gaussian Process Implementation

Kernel Function : SquaredExponentialKernel defines the covariance between any two data points $k(x_i, x_j)$.

```
class SquaredExponentialKernel:
    def __init__(self, length_scale=1.0, sigma_f=1.0):
        self.length_scale = length_scale
        self.sigma_f = sigma_f

    def get_params(self):
        return np.array([self.length_scale, self.sigma_f])

    def set_params(self, params):
        self.length_scale = params[0]
        self.sigma_f = params[1]

    def __call__(self, X1, X2):
        if X1.ndim == 1: X1 = X1[:, np.newaxis]
        if X2.ndim == 1: X2 = X2[:, np.newaxis]

        if X1.shape[1] != X2.shape[1]:
            raise ValueError(f"X1 and X2 must have the same number of features. Got {X1.shape[1]} and {X2.shape[1]}")

        sqdist = np.sum(X1**2, 1).reshape(-1, 1) + np.sum(X2**2, 1) - 2 * np.dot(X1, X2.T)
        sqdist = np.clip(sqdist, 0, np.inf) # Ensure non-negative for sqrt if used elsewhere

        return self.sigma_f**2 * np.exp(-0.5 / self.length_scale**2 * sqdist)
```

Listing 1: SquaredExponentialKernel from atgp.py

Covariance Matrix Calculation : This function applies given kernel to pairs of data points.

```
def calculate_covariance_matrix(X1, X2, kernel_func):
    return kernel_func(X1, X2)
```

Listing 2: Covariance matrix calculation snippet from atgp.py

General GP prediction : This function in SimpleGPR class implements it in the following way.

```
# In SimpleGPR class:
def predict(self, X_star):
    # ...
    K_star = self.kernel(X_star, self.X_train)
    mean_star = K_star @ self.alpha_

    K_star_star_diag = np.diag(self.kernel(X_star, X_star))
    # ... v = L^-1 @ K_star.T ...
    # ... var_reduction_diag = np.sum(v**2, axis=0) ...
    variance_star_diag = K_star_star_diag + (self._noise_std**2) -
    var_reduction_diag
    # ...
    return mean_star, variance_star_diag.reshape(-1,1)

```

Log Marginal Likelihood for Standard GP : This is the standard objective function maximized for hyperparameter tuning in a regular GP.

```
# In SimpleGPR class:
def log_marginal_likelihood(self, params_array_opt, X, y):
    # ... (setup kernel, noise) ...
    K = calculate_covariance_matrix(X, X, temp_kernel)
    K_noisy = K + current_noise_var * np.eye(N) + jitter * np.eye(N)
    # ... (Cholesky, solve for alpha_solve, log_det_K_noisy) ...
   lml = -0.5 * y.T @ alpha_solve - 0.5 * log_det_K_noisy - N/2.0 * np.log
    (2 * np.pi)
    # ...
    return lml.item()

```

3.2 ATGP code implementation

Conditional Distribution Optimization : The parameters which include kernel hyperparameters, b, mu, and noise terms are optimized by minimizing the negative of this log-likelihood in the ATGP.fit method.

```
# In ATGP class:
def log_marginal_likelihood_conditional(self, params_array_opt):
    # ... (parameter setup) ...
    current_length_scale = params_array_opt[0]
    current_sigma_f = params_array_opt[1]
    # ...
    current_b = params_array_opt[2]
    current_mu = params_array_opt[3]
    temp_lambda = self._get_lambda(b_param=current_b, mu_param=current_mu)
    # ...
    current_noise_S_var = params_array_opt[4]**2
    current_noise_T_var = params_array_opt[5]**2

    K11 = calculate_covariance_matrix(self.X_S, self.X_S, temp_kernel)
    K22 = calculate_covariance_matrix(self.X_T, self.X_T, temp_kernel)
    K_TS_base = calculate_covariance_matrix(self.X_T, self.X_S, temp_kernel
    )

    K21 = temp_lambda * K_TS_base
    K12 = K21.T

    # ... calculation of K11_noisy ...
    # mu_t = K21 @ inv(K11_noisy) @ y_S
```

```
    K11_noisy_inv_yS = scipy.linalg.solve_triangular(...)
    mu_t = K21 @ K11_noisy_inv_yS

    K11_noisy_inv_K12 = scipy.linalg.solve_triangular(...)
    C_t_main_term = K22 - K21 @ K11_noisy_inv_K12
    C_t = C_t_main_term + current_noise_T_var * np.eye(K22.shape[0])

    # ...
    log_likelihood = -0.5 * log_det_Ct - 0.5 * term2_quadratic - len(self.
    y_T)/2.0 * np.log(2 * np.pi)
    return log_likelihood.item()

def fit(self, method='L-BFGS-B', disp=False, maxiter=200):
    # ...
    objective = lambda params_opt: -self.
    log_marginal_likelihood_conditional(params_opt)
    result = minimize(objective, initial_params, method=method, bounds=
    bounds, ...)
    # ...

```

Semi-parametric Kernel & Gamma Distributed Rho : The value of λ is calculated and b and μ are optimized during fit.

```
        # In ATGP class:
def __init__(self, ..., initial_b=1.0, initial_mu=1.0, ...):
    # ...
    self._b = initial_b
    self._mu = initial_mu
    # ...

def _get_lambda(self, b_param=None, mu_param=None):
    b_to_use = b_param if b_param is not None else self._b
    mu_to_use = mu_param if mu_param is not None else self._mu
    # ... (parameter constraints) ...
    term_base = 1.0 / (1.0 + mu_to_use)
    # ... (error handling for pow_term) ...
    try:
        pow_term = np.power(term_base, b_to_use)
        lambda_val = 2 * pow_term - 1
    # ... (fallback) ...
    return np.clip(lambda_val, -1.0, 1.0)

# Used in log_marginal_likelihood_conditional:
# K21 = temp_lambda * K_TS_base

# Used in predict:
# K_tilde[:N_S, N_S:] = lambda_val * K_ST_base
# K_tilde[N_S:, :N_S] = lambda_val * K_ST_base.T
# k_x_star_rows = np.hstack((lambda_val * k_star_S_base, k_star_T_base))

```

Predict : This function calculates the final mean.

```
# In ATGP class:
def predict(self, X_star_T):
    # ...
    lambda_val = self._get_lambda()
    # ...
    y_all = np.vstack((self.y_S, self.y_T))

    # Construct K_tilde (combined covariance matrix for S and T training
    data)
```

```
    K_SS = calculate_covariance_matrix(self.X_S, self.X_S, self.base_kernel
    )
    K_TT = calculate_covariance_matrix(self.X_T, self.X_T, self.base_kernel
    )
    K_ST_base = calculate_covariance_matrix(self.X_S, self.X_T, self.
    base_kernel)

    K_tilde = np.zeros((N_all, N_all))
    K_tilde[:N_S, :N_S] = K_SS
    K_tilde[N_S:, N_S:] = K_TT
    K_tilde[:N_S, N_S:] = lambda_val * K_ST_base
    K_tilde[N_S:, :N_S] = lambda_val * K_ST_base.T

    Big_Lambda_diag = np.concatenate([np.full(N_S, noise_S_var_val),
                                      np.full(N_T, noise_T_var_val)])
    Big_Lambda = np.diag(Big_Lambda_diag)

    C_tilde = K_tilde + Big_Lambda + jitter * np.eye(N_all)
    # ... (alpha calculation)  ...

    k_star_S_base = calculate_covariance_matrix(X_star_T, self.X_S, self.
    base_kernel)
    k_star_T_base = calculate_covariance_matrix(X_star_T, self.X_T, self.
    base_kernel)
    # Apply lambda to the source part of k_x
    k_x_star_rows = np.hstack((lambda_val * k_star_S_base, k_star_T_base))

    mean_star = k_x_star_rows @ alph
    # ... (variance calculation) ...
    return mean_star, variance_star_diag.reshape(-1,1)

```

4 Experimental Setup and Results

Experiments are conducted on three datasets to evaluate the AT-GP model against baseline approaches.

4.1 Datasets

1. Wine Quality:
   (a) Source Task: Predicting quality of white wine (4898 samples).
   (b) Target Task: Predicting quality of red wine (1599 total samples, 5% for training).
2. WiFi:
   (a) Source Task: 500 samples
   (b) Target Task: 500 total samples, 5% for training.
3. SARCOS Robot Arm Inverse Dynamics:
   (a) Source Task: Predicting torque for Joint 1 (subsampled to 1000 samples).
   (b) Target Task: Predicting torque for Joint 2 (subsampled to 1000 total samples, 5% training)

4.2 Results Summary

The following table summarizes the Normalized Mean Squared Error (NMSE) achieved by this implementation. Lower is better.

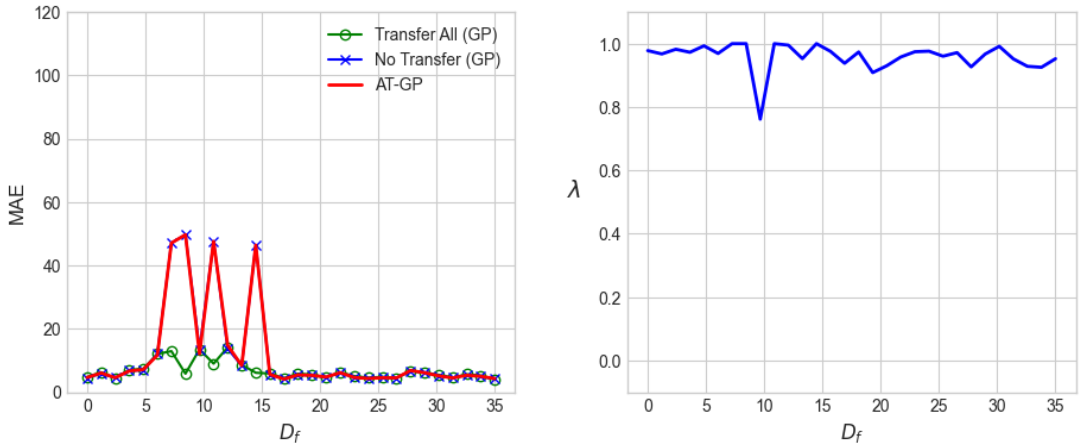

Figure 1: The left figure shows the change to MAE with increasing distance with f. The results are compared with transfer all and no transfer; The right figure shows the change to λ with increasing distance with f. (Own Implementation)

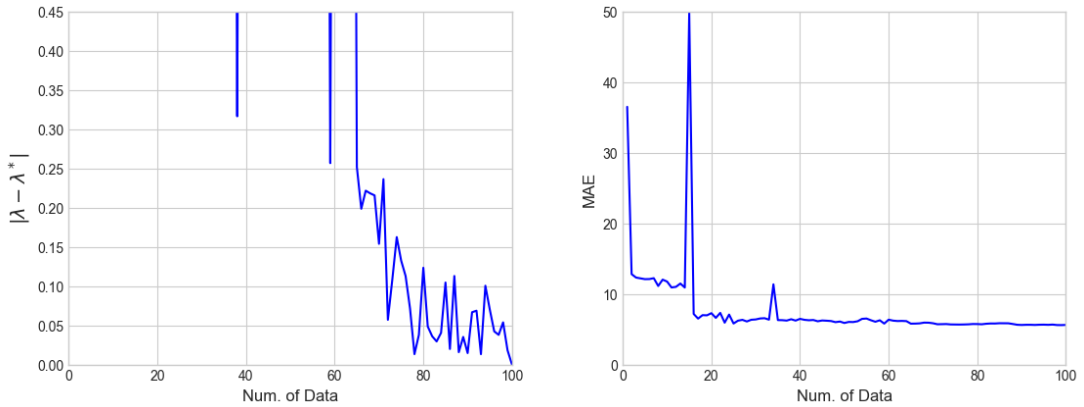

Figure 2: Learning with different numbers of labeled data in the target task. The left figure shows the convergence curve of λ with respect to the number of data. The right figure shows the change to MAE on test data. (Own Implementation)

Table 1: Results from this Implementation

Dataset	No Transfer GP	Transfer All GP	AT-GP
Wine	0.7379 (NMSE)	1.0631 (NMSE)	0.7345 (NMSE)
SARCOS	1.0206 (NMSE)	4.4362 (NMSE)	1.0150 (NMSE)
WiFi	13.3676 (Err Dist)	212.9710 (Err Dist)	13.2415 (Err Dist)

Note: Error Distance(in meters) is the metric for WiFi dataset whereas NMSE is the metric considered for Wine and SARCOS.

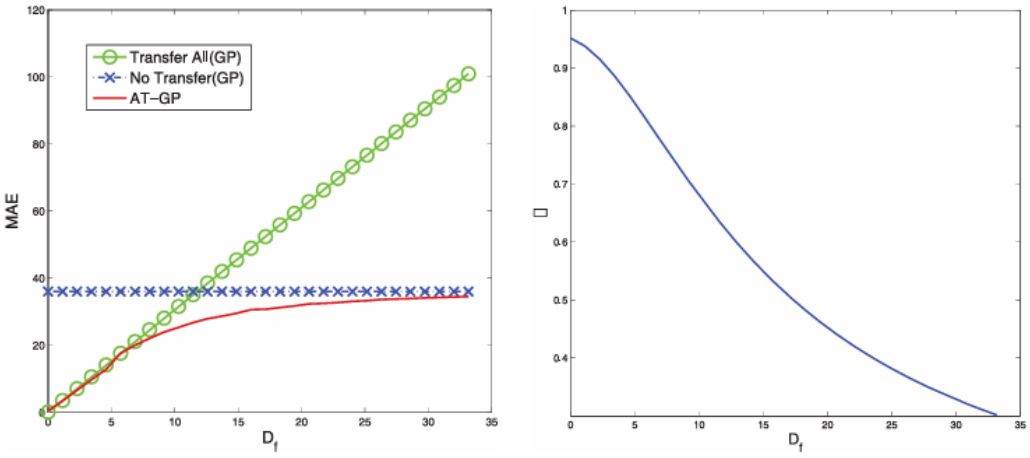

Figure 3: The left figure shows the change to MAE with increasing distance with f. The results are compared with transfer all and no transfer; The right figure shows the change to λ with increasing distance with f. (Original Implementation)

4.3 Comparison with Paper's Results

The paper (Cao et al., 2010, Table 1) reports the following mean results:

Table 2: Results from Cao et al. (2010, Table 1)

Data	Paper's No Transfer	Paper's Transfer All	Paper's AT-GP
Wine	1.33 (NMSE)	1.37 (NMSE)	1.16 (NMSE)
SARCOS	0.21 (NMSE)	1.58 (NMSE)	0.18 (NMSE)
WiFi	9.18 (Err Dist)	5.28 (Err Dist)	4.98 (Err Dist)

5 Conclusion

According to the method proposed in this paper, how much to transfer is based on how similar the tasks are and negative transfer can be avoided. The experiments on both synthetic and real-world datasets verify the effectiveness of the proposed model **relatively**.

6 Acknowledgments

I acknowledge the assistance of AI language models (ChatGPT and Gemini) during the preparation of this document and the accompanying code. The AI was utilized for tasks such as:

- **Refining**(Not generating from scratch) Python code for the AT-GP implementation, GP implementation is done by self.

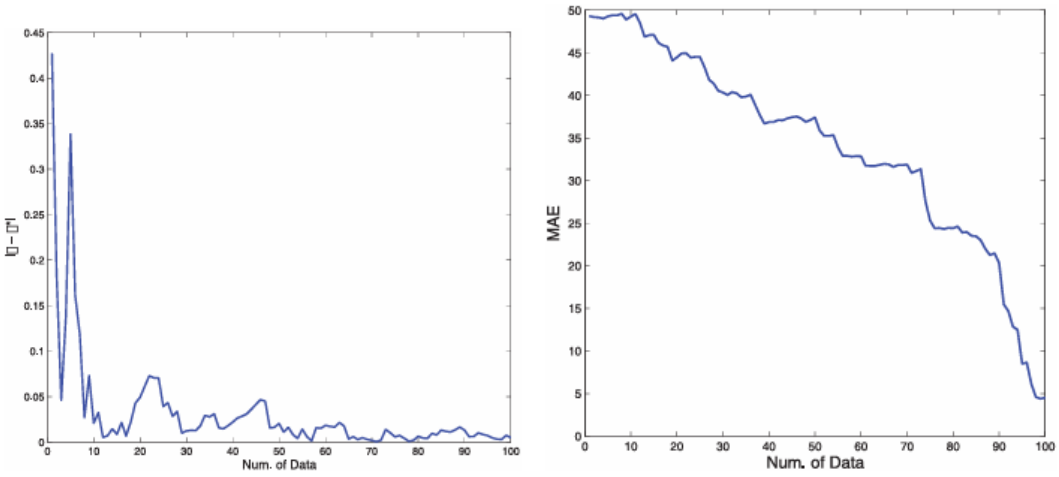

Figure 4: Learning with different numbers of labeled data in the target task. The left figure shows the convergence curve of λ with respect to the number of data. The right figure shows the change to MAE on test data. (Original Implementation)

- Modifying the code to generate the plots shown in this documentation (Code(at-gpr-AI-plotting.ipynb) is separately added in the zip folder).
- Assisting with LaTeX formatting and troubleshooting.