Other Models

Muchang Bahng

Spring 2025

Contents

Just some other models I've learned that don't fit in nicely to any of existing categories yet.

1 Topological Data Analysis

2 Geodesic Regression

In regression, note that we are finding a function $f : \mathcal{X} \to \mathcal{Y}$. In usual linear regression, both $\mathcal{X}, \mathcal{Y}$ are Euclidean space. However, there are cases where it may not be realistic that one or more of them should be modeled as a vector space. Rather, they may be part of a lower-dimensional manifold. For instance, if we want to use linear regression to predict the top k principal components of a dataset, they must be orthogonal, i.e. must be in a *Stiefl manifold*.

There are way to model this. For instance, we could have a projection operator that maps from $\mathbb{R}^m \to \mathcal{Y}$. This has its issues too, for one not being very efficient since perhaps the dimension of $\mathcal{Y}$ may be much less than m. Therefore, it might be better to directly regress onto a manifold. There were many attempts at this, but the first model to generalize OLS to manifolds was created by Fletcher in 2011 [Fle11] and expanded shortly in [TF13].

We start with the case where there is one covariate (i.e. $\mathcal{X} = \mathbb{R}$) and $\mathcal{Y} = (M, d)$ is a smooth Riemannian manifold with a metric. Recall that for a smooth manifold M, for any $p \in M$ and $v \in T_p M$, the tangent space at p, there is a unique geodesic curve $\gamma : [0, 1] \to M$ satisfying $\gamma(0) = p$, $\gamma'(0) = v$. This geodesic is guaranteed to exist locally, and with this,we can define the exponential map at p in the direction of v as

$$\exp_p(v) = \exp(p, v) = \gamma(1) \tag{1}$$

In other words, the exponential map takes a position and velocity as input and returns the point at time 1 along the geodesic with these initial conditions. With this motivation, we use slightly different notation than regular linear regression, referring p as the bias and v as the coefficient.

Definition 2.1 (Geodesic Regression)

The **geodesic regression** model is a probabilistic model that predicts the conditional distribution of $y \in (M, d)$ given $x \in \mathbb{R}$ as

$$y = \exp(\exp(p, vx), \epsilon) \tag{2}$$

where the parameters are $\theta = \{p, v\}$, and ϵ is a random variable defined over the tangent space at $\exp(p, vx)$.

Note that if we set $\mathcal{Y} = \mathbb{R}^m$, then we get the ordinary linear regression model back.

Definition 2.2 (Least Squares Geodesic Regression)

The least squares geodesic regression aims to minimize the MSE loss

$$L(\theta, (x, y)) = L(p, v, x, y) = d(\exp(p, vx), y)^2 \tag{3}$$

Lemma 2.1 (Risk)

The risk is

$$R(f) = \mathbb{E}_{x,y} \left[d(\exp(p, vx), y)^2 \right] \tag{4}$$

and the empirical risk for a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ is

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^n d(\exp(p, vx^{(i)}), y^{(i)})^2 \tag{5}$$

Unfortunately, minimizing this does not yield an analytic solution.

Example 2.1 (Code Walkthrough)

Let us fit a line onto this. We first define our manifold class with the matrix exponential and logarithm methods.

```
class S2:
  @staticmethod
  def exp(p, v):
    v_norm = np.linalg.norm(v)
    if v_norm < 1e-8:
      return p
    return np.cos(v_norm) * p + np.sin(v_norm) * v / v_norm

  @staticmethod
  def log(p, q):
    cos_dist = np.clip(np.dot(p, q), -1, 1)
    if np.abs(cos_dist - 1) < 1e-8:
      return np.zeros_like(p)

    theta = np.arccos(cos_dist)
    sin_theta = np.sin(theta)

    if sin_theta < 1e-8:
      return np.zeros_like(p)

    return theta * (q - cos_dist * p) / sin_theta

  @staticmethod
  def distance(p, q):
    cos_dist = np.clip(np.dot(p, q), -1, 1)
    return np.arccos(cos_dist)

  @staticmethod
  def project_to_tangent(p, v):
    return v - np.dot(v, p) * p

  @staticmethod
  def normalize(x):
    return x / np.linalg.norm(x)
```

Next, we define our data generation process.

```
def generate_sample_data(n_samples=50, noise_level=0.1):
  X = np.random.uniform(-2, 2, n_samples)

  p_true = S2.normalize(np.array([1, 0, 0]))
  v_true = S2.normalize(np.array([0, 1, 0.2]))
  v_true = S2.project_to_tangent(p_true, v_true) * 0.5

  Y = []
  for x in X:
    y_clean = S2.exp(p_true, v_true * x)

    noise = np.random.normal(0, noise_level, 3)
    noise = S2.project_to_tangent(y_clean, noise)
    y_noisy = S2.exp(y_clean, noise)
    y_noisy = S2.normalize(y_noisy)
```

```

    Y.append(y_noisy)

  return X, np.array(Y), p_true, v_true
```

Finally, we define our regression model and optimize the loss with BFGS (SGD doesn't work very well here).

```
class GeodesicRegression:
  def __init__(self):
    self.p = None
    self.v = None

  def _geodesic_point(self, p, v, x):
    return S2.exp(p, v * x)

  def _objective(self, params, X, Y):
    p_flat = params[:3]
    v_flat = params[3:6]

    p_flat = S2.normalize(p_flat)
    v_flat = S2.project_to_tangent(p_flat, v_flat)

    total_loss = 0.0
    for i in range(len(X)):
      pred = self._geodesic_point(p_flat, v_flat, X[i])
      loss = S2.distance(pred, Y[i])**2
      total_loss += loss

    return total_loss / len(X)

  def fit(self, X, Y, p_init=None, v_init=None, method='BFGS'):
    X = np.array(X)
    Y = np.array(Y)

    if p_init is None:
      p_init = S2.normalize(np.array([1, 0, 0]))
    if v_init is None:
      v_init = np.array([0, 0.1, 0])

    v_init = S2.project_to_tangent(p_init, v_init)

    initial_params = np.concatenate([p_init, v_init])

    result = minimize(
      self._objective,
      initial_params,
      args=(X, Y),
      method=method,
      options={'disp': False}
    )

    if result.success:
      self.p = S2.normalize(result.x[:3])
      self.v = S2.project_to_tangent(self.p, result.x[3:6])
      return result
    else:
```

```
            raise RuntimeError(f"Optimization failed: {result.message}")

    def predict(self, X):
        X = np.array(X)
        predictions = []

        for x in X:
            pred = self._geodesic_point(self.p, self.v, x)
            predictions.append(pred)

        return np.array(predictions)

    def score(self, X, Y):
        predictions = self.predict(X)
        total_loss = 0.0

        for i in range(len(Y)):
            loss = S2.distance(predictions[i], Y[i])**2
            total_loss += loss

        return total_loss / len(Y)

X_train, Y_train, p_true, v_true = generate_sample_data(n_samples=30)
model = GeodesicRegression()
result = model.fit(X_train, Y_train)
```

This gives the following, which is a good estimate of the original parameters.

$$\hat{p} = \begin{pmatrix} 0.99993911 \\ -0.01038634 \\ -0.00372747 \end{pmatrix} \approx \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = p_{\text{true}}, \qquad \hat{v} = \begin{pmatrix} 0.00530609 \\ 0.48332324 \\ 0.07667702 \end{pmatrix} \approx \begin{pmatrix} 0 \\ 0.49029034 \\ 0.09805807 \end{pmatrix} = v_{\text{true}} \tag{6}$$

The following figure also visualizes this.

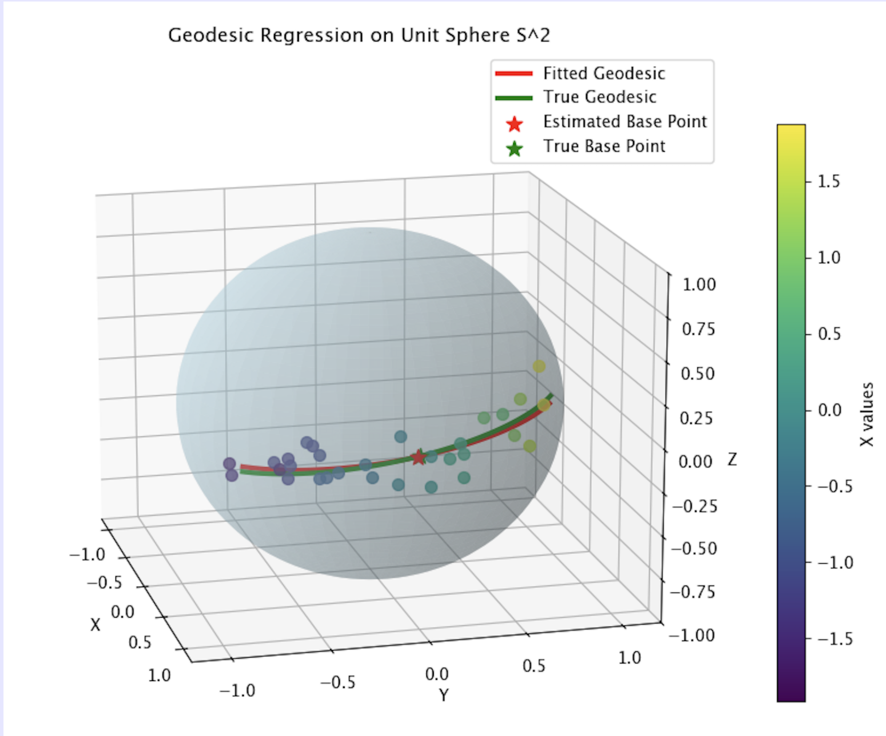

Figure 1

2.1 Multiple Geodesic Regression

Note that this was a model for a path in some manifold, and naturally we would like to extend this to have multiple covariates. Kim in 2014 did exactly that, and provided a framework for multivariate general linear models in [KAC+14].

Definition 2.3 (Multiple Geodesic Regression)

The **multiple geodesic regression** model is a probabilistic model that predicts the conditional distribution of $y \in (M, d)$ given $x \in \mathbb{R}$ as

$$y = \exp\left(\exp\left(p, \sum_{i=1}^{d} x_j v_j\right), \epsilon\right) = \exp\left(\exp(p, Vx)\right) \tag{7}$$

where the parameters are $\theta = \{p \in \mathbb{R}^m, V \in \mathbb{R}^{m \times d}\}$, and ϵ is a random variable defined over the tangent space at $\exp(p, Vx)$.

Definition 2.4 (Least Squares Geodesic Regression)

The least squares geodesic regression aims to minimize the MSE loss

$$L(\theta, (x, y)) = L(p, v, x, y) = d(\exp(p, Vx), y)^2 \tag{8}$$

Example 2.2 (Code Walkthrough)

We demonstrate this by conducting geodesic regression on a dataset of 50 samples ($x \in \mathbb{R}^2, y \in S^2$). We start by generating a 2-dimensional toy dataset according to our model.

```python
def generate_sample_data(n_samples=50, n_features=2, noise_level=0.1):
  X = np.random.uniform(-1, 1, (n_samples, n_features))

  p_true = S2.normalize(np.array([1, 0, 0]))
  V_true = np.array([[0, 0.3], [0.5, -0.2], [0.2, 0.4]])

  for i in range(n_features):
    V_true[:, i] = S2.project_to_tangent(p_true, V_true[:, i])

  Y = []
  for x in X:
    tangent_vec = V_true @ x
    y_clean = S2.exp(p_true, tangent_vec)

    noise = np.random.normal(0, noise_level, 3)
    noise = S2.project_to_tangent(y_clean, noise)
    y_noisy = S2.exp(y_clean, noise)
    y_noisy = S2.normalize(y_noisy)

    Y.append(y_noisy)

  return X, np.array(Y), p_true, V_true
```

```python
class MultipleGeodesicRegression:
  def __init__(self, n_features):
    self.n_features = n_features
    self.p = None
    self.V = None

  def _geodesic_point(self, p, V, x):
    tangent_vec = V @ x
    return S2.exp(p, tangent_vec)

  def _objective(self, params, X, Y):
    p_flat = params[:3]
    V_flat = params[3:].reshape(3, self.n_features)

    p_flat = S2.normalize(p_flat)

    for i in range(self.n_features):
      V_flat[:, i] = S2.project_to_tangent(p_flat, V_flat[:, i])

    total_loss = 0.0
    for i in range(len(X)):
      pred = self._geodesic_point(p_flat, V_flat, X[i])
      loss = S2.distance(pred, Y[i])**2
      total_loss += loss

    return total_loss / len(X)

  def fit(self, X, Y, p_init=None, V_init=None, method='BFGS'):
```

```
    X = np.array(X)
    Y = np.array(Y)

    if p_init is None:
      p_init = S2.normalize(np.array([1, 0, 0]))
    if V_init is None:
      V_init = np.random.normal(0, 0.1, (3, self.n_features))

    for i in range(self.n_features):
      V_init[:, i] = S2.project_to_tangent(p_init, V_init[:, i])

    initial_params = np.concatenate([p_init, V_init.flatten()])

    result = minimize(
      self._objective,
      initial_params,
      args=(X, Y),
      method=method,
      options={'disp': False}
    )

    if result.success:
      self.p = S2.normalize(result.x[:3])
      self.V = result.x[3:].reshape(3, self.n_features)

      for i in range(self.n_features):
        self.V[:, i] = S2.project_to_tangent(self.p, self.V[:, i])

      return result
    else:
      raise RuntimeError(f"Optimization failed: {result.message}")

  def predict(self, X):
    X = np.array(X)
    predictions = []

    for x in X:
      pred = self._geodesic_point(self.p, self.V, x)
      predictions.append(pred)

    return np.array(predictions)

  def score(self, X, Y):
    predictions = self.predict(X)
    total_loss = 0.0

    for i in range(len(Y)):
      loss = S2.distance(predictions[i], Y[i])**2
      total_loss += loss

    return total_loss / len(Y)
```

The results show that it is a good estimate. Both the initial point $\hat{p}$ and the matrix $\hat{V}$ are good estimators.

$$\hat{p} = \begin{pmatrix} 0.99984466 \\ 0.01430395 \\ 0.01029851 \end{pmatrix} \approx \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = p, \qquad \hat{V} = \begin{pmatrix} -0.00905151 & -0.00171887 \\ 0.48494844 & -0.18346139 \\ 0.20521662 & 0.42169444 \end{pmatrix} \approx \begin{pmatrix} 0 & 0 \\ 0.5 & -0.2 \\ 0.2 & 0.4 \end{pmatrix} = V \tag{9}$$

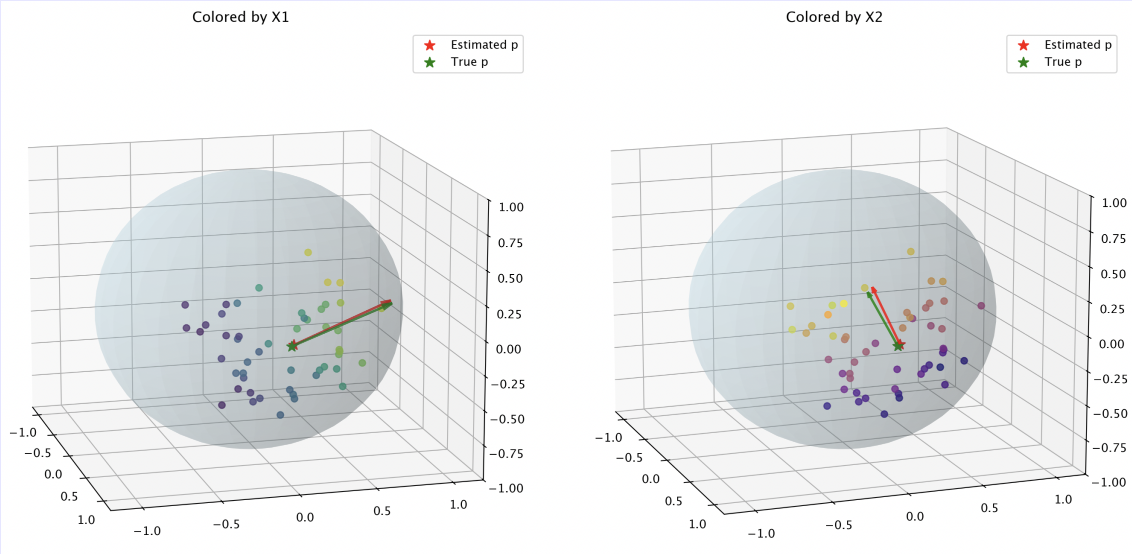

Figure 2: The estimated values of the first column of V (left) and the second column of V (right) are good approximations of the true. Note that they point in the direction of the gradients.

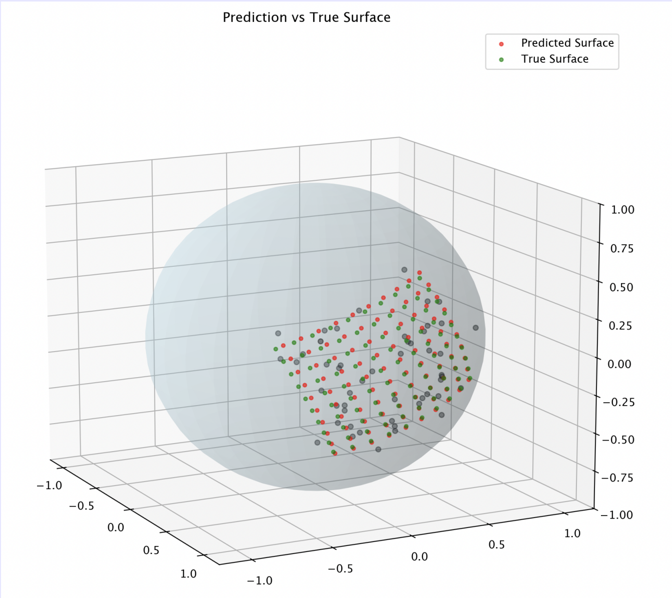

Figure 3: Since we are regressing a 2-dimensional space onto a 2-dimensional manifold, the image of f is trivially S^2 except in degenerate cases. However, it is still nice to compare our predicted values $\hat{y} = f(x)$ to the true y.

2.2 Robust Geodesic Regression

3 Frechet Regression

Frechet regression generalizes regression on manifolds to general metric spaces (with some probability measure) [PM17]. In linear regression, we try to estimate the conditional distribution $g(x) = \mathbb{E}[Y \mid X = x]$ with some linear function.

$$y = \beta^T x = \beta_0 + \beta_1 x_1 + \ldots + \beta_d x_d \tag{10}$$

Note that this requires

Definition 3.1 (Frechet Mean)

The **Frechet mean** of a set of points $y^{(1)}, \ldots, y^{(n)} \in (M, d)$ in a metric space is

$$\mu = \operatorname*{argmin}_{m \in M} \sum_{i=1}^{n} d^2(y^{(i)}, m) \tag{11}$$

if a unique value exists.

Now we can define the conditional Frechet mean. Since M is a probability space, we can define the integral with respect to the conditional distribution $\mathbb{P}(Y \mid X = x)$. The problem is that to compute the integral $\int f(y)\, d\mathbb{P}(Y \mid x)$, we need f to be in some vector space—where addition and scalar multiplication are defined. We can take the Frechet mean to be this function, where the argmin is taken outside the integral.

Definition 3.2 (Conditional Frechet Mean)

The **conditional Frechet mean** is defined

$$\mu(x) = \operatorname*{argmin}_{m \in M} \mathbb{E}_y \left[d^2(y, m) \mid X = x \right] \tag{12}$$

All that is left to do is try to represent this function μ with some parametric model.

References

[Fle11] Thomas Fletcher. Geodesic Regression on Riemannian Manifolds. In Pennec, Xavier, Joshi, Sarang, Nielsen, and Mads, editors, *Proceedings of the Third International Workshop on Mathematical Foundations of Computational Anatomy - Geometrical and Statistical Methods for Modelling Biological Shape Variability*, pages 75–86, Toronto, Canada, September 2011.

[KAC+14] Hyunwoo J. Kim, Nagesh Adluru, Maxwell D. Collins, Moo K. Chung, Barbara B. Bendin, Sterling C. Johnson, Richard J. Davidson, and Vikas Singh. Multivariate general linear models (mglm) on riemannian manifolds with applications to statistical analysis of diffusion weighted images. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2705–2712, 2014.

[PM17] Alexander Petersen and Hans-Georg Müller. Fréchet regression for random objects with euclidean predictors, 2017.

[TF13] P. Thomas Fletcher. Geodesic regression and the theory of least squares on riemannian manifolds. *Int. J. Comput. Vision*, 105(2):171–185, November 2013.