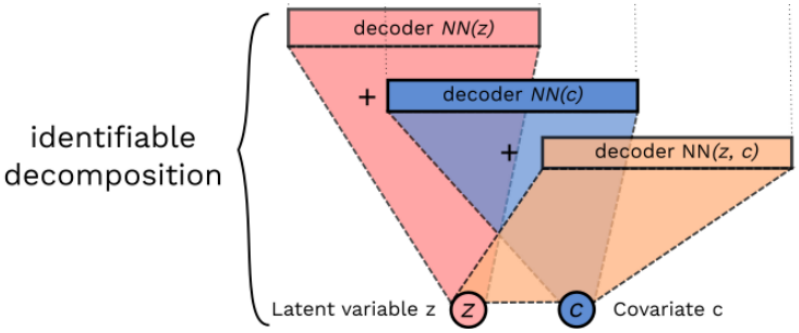
Neural Decomposition: Functional ANOVA with Variational Autoencoders

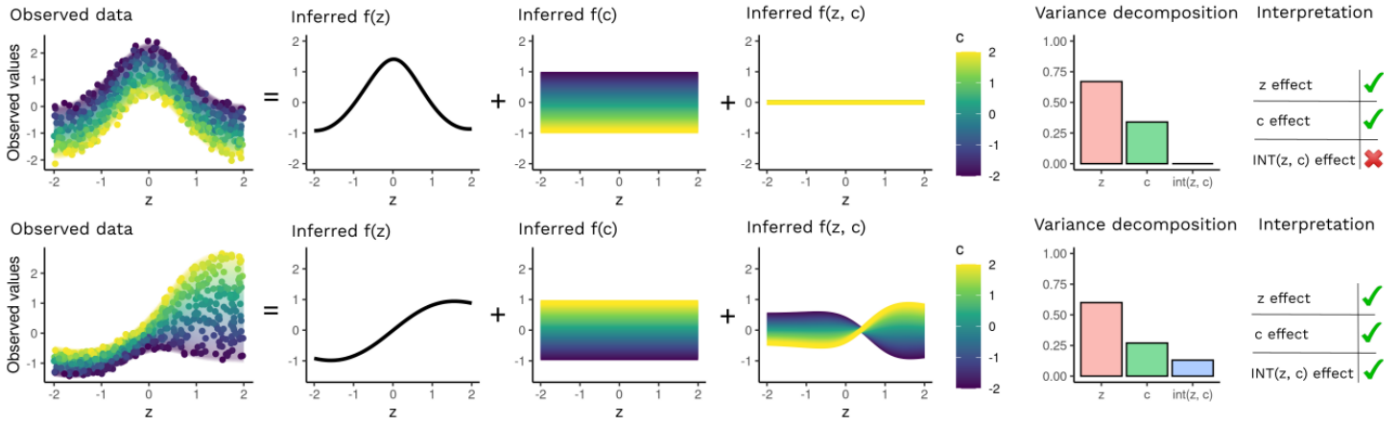
(Märtens and Yau 2020)

Neural decomposition step-by-step

Adaptation of functional ANOVA

Aim: **feature-level interpretability** that would let characterise the sources of variation for individual features





Adaptation of functional ANOVA

Aim: **feature-level interpretability** that would let characterise the sources of variation for individual features

$\rightarrow$ **functional ANOVA decomposition** of the decoder network $f^{\theta}(z_i, c_i)$ to extract additive marginal and interaction effects

$$f^{\theta}(z_i, c_i) = f_0^{\theta} + f_z^{\theta}(z_i) + f_c^{\theta}(c_i) + f_{zc}^{\theta}(z_i, c_i)$$

with latent and fixed inputs collectively denoted $x := (z, c)$ can be generalized as

$$f_0^{\theta} + \sum_k f_k^{\theta}(x_{ik}) + \sum_{k,l} f_{kl}^{\theta}(x_k, x_l) + \ldots + f_{1,\ldots D}^{\theta}(x)$$

(cf. appendix slides)

Need for constraints

- without additional constraints, the decomposition is **unidentifiable**
  - the functional subspaces f_I can be seen as functions defined on the same input space, being constant in the rest of coordinates, and these **subspaces are overlapping** ($f_I^\theta(x_I)$ is a subset of $f_{I,J}^\theta(x_I, x_J)$)

$\rightarrow$ with no further constraints, no interpretation since **higher order terms can absorb the variability** that could be explained by main effects or lower-order interactions

Integral constraints

- as for functional ANOVA, **integral constraints** to turn this into a identifiable learning problem
- **constrain the marginal effects** of every neural network $f_I^\theta(x)$ to be zero:

$$\int f_I(x_I) dx_i = 0 \; for \; all \; i \in I$$

- direct consequences of the integral constraints:

$\rightarrow$ the functional subsbaces corresponding to f_I^θ and $f_{I,J}^\theta$ **do not overlap** anymore

$\rightarrow$ these functional subspaces are **orthogonal** in L_2

Variance decomposition

- no overlap = **identifiability**
- orthogonality = **interpretable variance decomposition**

e.g. for a 2D input (x_1, x_2)

- the decomposition is f_0^θ, f_1^θ, f_2^θ, f_{12}^θ
- with the corresponding integral constraints:
  - $\int f_1^\theta(x_1)dx_1 = 0$
  - $\int f_2^\theta(x_2)dx_2 = 0$
  - $\int f_{12}^\theta(x_1, x_2)dx_2 = 0$ for all x_1
  - $\int f_{12}^\theta(x_1, x_2)dx_1 = 0$ for all x_2

Variance decomposition

similarly for (c, z)

- the decomposition is f_0^θ, f_c^θ, f_z^θ, f_{cz}^θ
- with the corresponding integral constraints:
  - $\int f_c^\theta(c) dx_c = 0$
  - $\int f_z^\theta(z) dx_z = 0$
  - $\int f_{cz}^\theta(c, z) dz = 0$ for all c
  - $\int f_{cz}^\theta(c, z) dc = 0$ for all z

How to do inference with integral constraints ?

= **constrained optimization problem**

- could be solved with:
  1. penalty method
  2. Augmented Lagrangian method (aka BDMM, method of multipliers) (adapted to neural networks by Platt and Barr 1988)
  3. hybrid multiplier + penalty method (aka MDMM or HDMM, also from Platt and Barr 1988)

  (will be exemplified for 1D scenario)

(cf. appendix slides)

Optimization: 1D scenario - "loss design"

- for a decoder $f^{\theta}(x)$: optimize the ELBO with the constraint $\int f^{\theta}(x)dx = 0$

(= restrict f^{θ} to a subspace such that $\int f^{\theta}(x)dx = 0$)

$\rightarrow$ **augment the ELBO with additional penalty term(s)** which will be $= 0$ when the constraints are fulfilled during optimization

Optimization: 1D scenario - penalty method 1

How to add penalty to the ELBO ?

1. **penalty method**, with fixed c

$$c(\int f^{\theta}(x)dx)^2$$

- *downside*: no guarantees that the constraints will be fulfilled (no convergence)

Optimization: 1D scenario - penalty method 2

How to add penalty to the ELBO ?

2. **BDMM**: add a penalty which is treated as a parameter (analogous to the use of Lagrange multipliers)

$$\lambda \int f^{\theta}(x)dx$$

- optimize NN parameters as usual with **gradient descent** $\lambda^{t+1} = \lambda^{t} - \eta(\int f^{\theta}(x)dx)$
- optimize λ simultaneously with **gradient ascent** $\lambda^{t+1} = \lambda^{t} + \eta(\int f^{\theta}(x)dx)$ (η = learning rate)
- *downside*: converge to zero, but with damped oscillation behaviour (slow convergence)

Optimization: 1D scenario - penalty method 3

How to add penalty to the ELBO ?

3. combine the 2 penalty terms in a hybrid constrained optimization objective (**HDMM**)

$$\min_{\theta,\phi}\{-L^{\theta,\phi} + \lambda \int f^{\theta}(x)dx + c(\int f^{\theta}(x)dx)^2\}$$

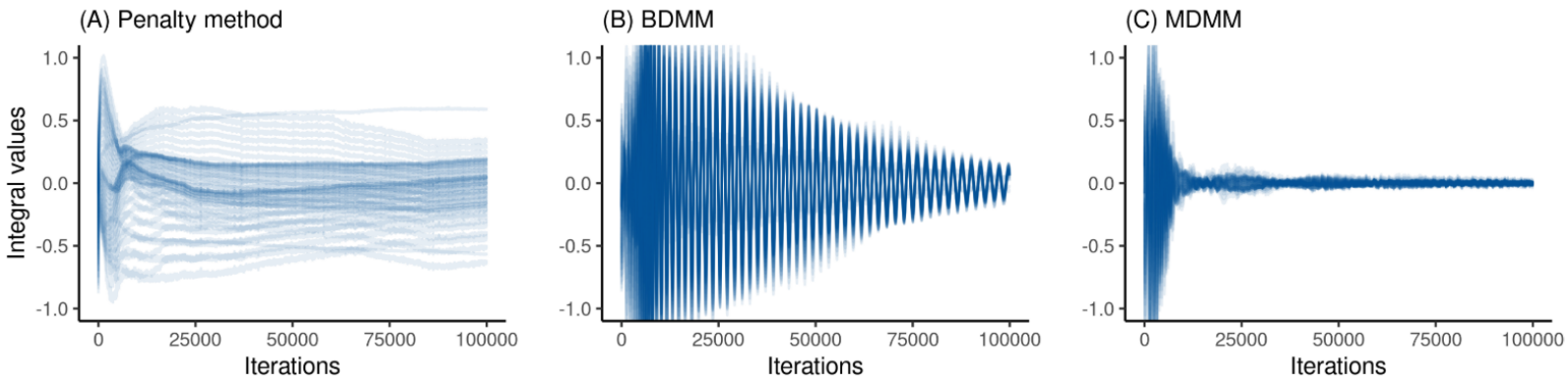
- c is constant, λ is optimized and L is the ELBO of the VAE
- this scheme corresponds to the MDMM of Platt and Barr 1988
- empirical observation that replacing fixed c with a sequence of $c^1 \leq \ldots \leq c^T$ leads to faster convergence

Optimization with HDMM: 2D scenario

e.g. for a 2D input (x_1, x_2)

- functional decomposition: f_0^θ, f_1^θ, f_2^θ, f_{12}^θ
- integral constraints: $\int f_1^\theta(x_1)dx_1 = 0$, $\int f_2^\theta(x_2)dx_2 = 0$, $\int f_{12}^\theta(x_1, x_2)dx_2 = 0$ for all x_1, $\int f_{12}^\theta(x_1, x_2)dx_1 = 0$ for all x_2
- corresponding penalty terms:
  - $\lambda_1 \int f_1^\theta(x_1)$
  - $\lambda_2 \int f_2^\theta(x_2)$
  - $\lambda_1(x_1)(\int f_{12}^\theta(x_1, x_2)dx_1)dx_2$ for every x_1
    (introduced LM $\lambda_1(x_1)$ indexed by a continuous-valued x_1)
  - $\lambda_2(x_2)(\int f_{12}^\theta(x_1, x_2)dx_2)dx_1$ for every x_2
    (introduced LM $\lambda_2(x_2)$ indexed by a continuous-valued x_2)

Comparison: penalty / BDMM / HDMM



Traces for $\int f^{\theta}(x_1, x_2)dx_2$ *on a grid of* x_1 *values (each line corresponds to one grid point) over 100 000 iterations.*

- *(A) the constraints have not been fulfilled by the penalty method with a fixed* c*.*
- *(B) BDMM exhibits oscillating behaviour and integrals are slowly converging towards zero.*
- *(C) MDMM leads to optimisation which results in* ≈ 0 *integral values much more quickly.*

Optimization with HDMM: 2D scenario

Similarly for (c, z) input we can write:

- functional decomposition: f_0^θ, f_c^θ, f_z^θ, f_{cz}^θ
- integral constraints: $\int f_c^\theta(x_c)dx_c = 0$, $\int f_z^\theta(x_z)dx_z = 0$, $\int f_{cz}^\theta(c, z)dx_z = 0$ for all x_c, $\int f_{cz}^\theta(c, z)dx_c = 0$ for all z
- corresponding penalty terms:
  - $\lambda_c \int f_c^\theta(c)$
  - $\lambda_z \int f_z^\theta(z)$
  - $\lambda_c(c)(\int f_{cz}^\theta(c, z)dx_c)dx_z$ for every c
  - $\lambda_z(z)(\int f_{cz}^\theta(c, z)dx_z)dx_c$ for every z

Optimization in the code (2D): penalty & loss (decoder)

```
def calculate_penalty(self): // from decoder.py
    int_z, int_c, int_cz_1, int_cz_2 = self.calculate_integrals()

    # penalty with fixed lambda0
    if self.penalty_type in ["fixed", "MDMM"]:
        penalty0 = self.lambda0 * (int_z.abs().mean() + int_c.abs().mean() + \
                                   int_cz_1.abs().mean() + int_cz_2.abs().mean())

    if self.penalty_type in ["BDMM", "MDMM"]:
        penalty_BDMM = (self.Lambda_z * int_z).mean() + (self.Lambda_c * int_c).mean() + \
                       (self.Lambda_cz_1 * int_cz_1).mean() + (self.Lambda_cz_2 * int_cz_2).mean()

    if self.penalty_type == "fixed":
        penalty = penalty0
    elif self.penalty_type == "BDMM":
        penalty = penalty_BDMM
    elif self.penalty_type == "MDMM":
        penalty = penalty_BDMM + penalty0
    [...]

def loss(self, y_pred, y_obs): // from decoder.py
    [...]
    total_loss = - self.loglik(y_pred, y_obs) + penalty
    [...]
```

Optimization in the code (2D): total loss

```
def forward(self, data_subset, beta=1.0, device="cpu"): // in CVAE.py
    [...]
    # encode
    mu_z, sigma_z = self.encoder(Y, c)
    eps = torch.randn_like(mu_z)
    z = mu_z + sigma_z * eps

    # decode
    y_pred = self.decoder.forward(z, c)
    decoder_loss, [...] = self.decoder.loss(y_pred, Y)

    # loss function
    VAE_KL_loss = KL_standard_normal(mu_z, sigma_z)

    total_loss = decoder_loss + beta * VAE_KL_loss
```

Optimization in the code (2D): λ optimization

```
// in CVAE.py
def optimize:
   [...]
   self.decoder.Lambda_z += augmented_lagrangian_lr * int_z
   self.decoder.Lambda_c += augmented_lagrangian_lr * int_c
   self.decoder.Lambda_cz_1 += augmented_lagrangian_lr * int_cz_dc
   self.decoder.Lambda_cz_2 += augmented_lagrangian_lr * int_cz_dz
   [...]
```

Integral calculation

integrals can be estimated using either quadrature or Monte Carlo estimates

```
def calculate_integrals(self): // from decoder.py
    # has shape [1, output_dim]
    int_z = self.forward_z(self.grid_z).mean(dim=0).reshape(1, self.output_dim)

    # has shape [1, output_dim]
    int_c = self.forward_c(self.grid_c).mean(dim=0).reshape(1, self.output_dim)

    m1 = self.n_grid_z
    m2 = self.n_grid_c
    out = self.forward_cz_concat(self.grid_cz)
    out = out.reshape(m1, m2, self.output_dim)
    # has shape [m1, output_dim]
    int_cz_dc = out.mean(dim=1)
    # has shape [m2, output_dim]
    int_cz_dz = out.mean(dim=0)

// in main script:
grid_z = torch.linspace(-2.0, 2.0, steps=grid_nsteps).reshape(-1, 1).to(device)
grid_c = torch.linspace(-2.0, 2.0, steps=grid_nsteps).reshape(-1, 1).to(device)
grid_cz = torch.cat(expand_grid(grid_z, grid_c), dim=1).to(device)
```

When have the constraints been satisfied ?

Proposed approach:

- establish a desired **tolerance threshold** ε
- evaluate the integrals after optimisation
- make sure that all NNs have been constrained to the desired functional subspaces within the desired tolerance

My issues/questions

- not sure about the functional/variance decomposition: if 3 variables, how will the interaction terms look like ?
- could we do functional/variance decomposition and ignore some (high-order interaction) terms ? (cf. christophm.github)
- how integrals are calculated:
  - not understood how does the grid story work...
  - how will it be for 2D latent space ? (how to adapt grid for 3D interaction term ? how will it work for higher-order interactions ?)
- not found in the code the "control" of convergence with ϵ and the sequence of c
- $+$ a series of questions about some mathematical aspects

some thoughts:

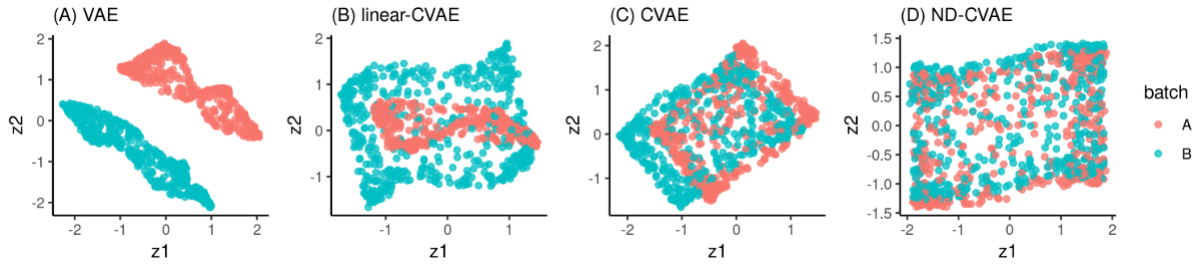
- if grids are problematic $\rightarrow$ how will it be to use instead a Monte Carlo procedure (as mentioned in the article) ?
- during decomposition, could we ignore the terms we are not interested in ?
- "Bayesian Functional ANOVA Modeling Using Gaussian Process Prior Distributions" by Kaufman and Sain (2010)
  - how about combining a Bayesian way knowledge injection with variance decomposition for neural networks ?
- forget about neural decomposition and find other ways to get feature-level interpretability...

Appendix: ND with 2 LDs 1/3

synthetic experiment generated from a two-dimensional latent space ($z1$, $z2$): 2 batches where each feature is either

- unperturbed,
- differs by a constant by batch or
- varies with $z1$ by batch.

The goal was to identify if any tested VAE variant was capable of achieving batch correction (here $c = batch$) by identifying a latent space in which the 2 batches overlapped each other.

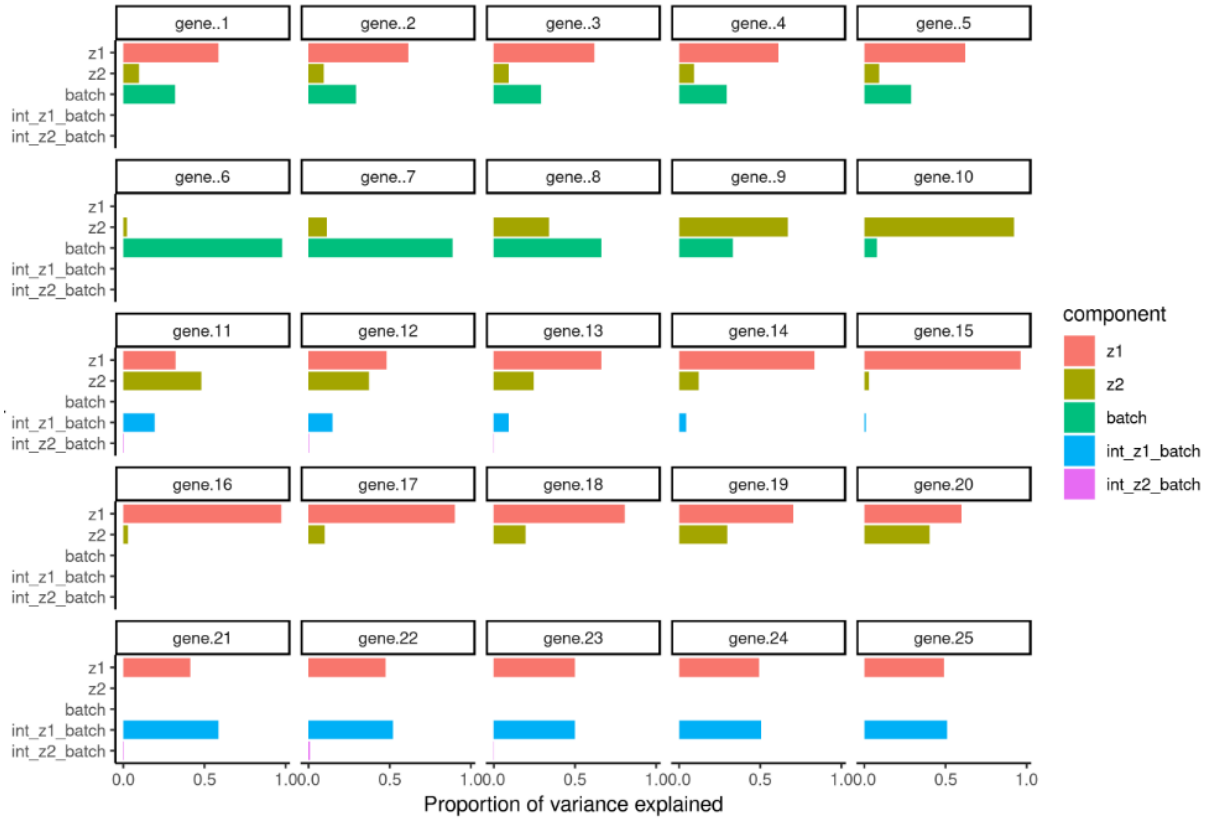


ND-CVAE recovers a batch-adjusted two-dimensional z on a synthetic example whereas other approaches (VAE, CVAE, and linear-CVAE) struggle to appropriately adjust for known c.

$\rightarrow$ the standard CVAE did not entirely remove the batch effect in the latent space
$\rightarrow$ the sparse ND structure within the ND-CVAE has correctly identified a $(z1, z2)$ space in which the batches are now intermixed and the nonlinear batch effects removed

Appendix: ND with 2 LDs 2/3

ND-CVAE lets characterise how features vary with latent $z1$, $z2$ and known c



On the synthetic batch-correction example, we characterised the decomposition learned by ND for every gene as a function of $z1$, $z2$, batch c, and interactions between them.

Appendix: ND with 2 LDs 3/3

```
# define encoder which maps (data, covariate) -> (z_mu, z_sigma)
encoder_mapping = nn.Sequential(
    nn.Linear(data_dim + n_covariates, hidden_dim),
    nn.ReLU(),
    nn.Linear(hidden_dim, 2)
)

encoder = cEncoder(z_dim=1, mapping=encoder_mapping)

decoder_z = nn.Sequential(
    nn.Linear(1, hidden_dim),
    nn.Tanh(),
    nn.Linear(hidden_dim, data_dim)
)

decoder_c = nn.Sequential(
    nn.Linear(1, hidden_dim),
    nn.Tanh(),
    nn.Linear(hidden_dim, data_dim)
)

decoder_cz = nn.Sequential(
    nn.Linear(2, hidden_dim),
    nn.Tanh(),
    nn.Linear(hidden_dim, data_dim)
)

decoder = Decoder(data_dim,
                  grid_z, grid_c, grid_cz,
                  decoder_z, decoder_c, decoder_cz,
                  has_feature_level_sparsity=True, p1=0.1, p2=0.1, p3=0.1,
                  lambda0=1e2, penalty_type="MDMM",
                  device=device)

decoder = Decoder(data_dim,
                grid_z1=grid_z1, grid_z2=grid_z2, grid_c=grid_c,
                grid_cz1=grid_cz1, grid_cz2=grid_cz2, grid_z1z2=grid_z1z2,
                mapping_z1=decoder_z1,  mapping_z2=decoder_z2, mapping_c=decoder_c,
                mapping_cz1=decoder_cz1, mapping_cz2=decoder_cz2,
                mapping_z1z2=decoder_z1z2,
                  has_feature_level_sparsity=True,
                  p1=0.1, p2=0.1, p3=0.1, p4=0.1,
                  p5=0.1, p6=0.1, p7=0.1,
                  lambda0=1e2, penalty_type="MDMM",
                  device=device)

# Combine the encoder + decoder and fit the decomposable CVAE
model = CVAE(encoder, decoder, lr=5e-3, device=device)
```

Appendix: functional decomposition 1/2

- functional decomposition takes this high-dimensional function and splits it into lower-dimensional components
  - allows to attribute effects to individual features and to identify interactions between features
- e.g. for a 2D function f

$$f(x_1, x_2) = f_0 + f_1(x_1) + f_2(x_2) + f_{1,2}(x_{1,2})$$

- f_0 is the **intercept**
  - what the prediction is when all feature effects are set to 0
- f_1 and f_2 are the **main effects** of x_1 and x_2
  - how each feature affects the prediction, independent of the values the other feature takes on
- $f_{1,2}$ is the **interaction effect** between the two features
  - what the effect of the features is together

Appendix: functional decomposition 2/2

- the **components themselves are functions** (except for the intercept) with different input dimensionalities
- a function that takes in a p-dimensional vector can be split into 2^p **components**
- no unique solution to functional decomposition
- we can move an effect between main effect and a higher interaction while the total prediction remains intact
  - the decomposition is **arbitrary** if we don't pose any limitations on how each of the components look like
- how to prevent ambiguity and compute the components ?
  $\rightarrow$ one way is **functional ANOVA**

Appendix: functional ANOVA (1/2)

- proposed by Hooker 2004
- estimate the individual components as:

$$f_S(x) = \int_{X_{-S}} (f(x))\, dX_{-S}) - \int_{X_{-S}} \left(\sum_{V \subset S} \right) dX_{-S})$$

- the first part is the **integral over the prediction function, with respect of the features that are not in the set**
  - this the same as the expectation of the function when we integrate out features X_{-S}, and pretending that all features follow a uniform distribution
- the second part are **all the lower dimensional components**, so we apply some kind of centering

Appendix: functional ANOVA (2/2)

- each higher order effect is defined by integrating over all other features, but also be removing all the lower-order effects that are subsets of the higher-order effects
- this fullfills a few desirable axioms:
  1. Zero Means: $\int f_S(x_S) dX_s = 0$ for each $S \neq \emptyset$.
     $\rightarrow$ **all effects or interactions are centered around zero**.
  2. Orthogonality: $\int f_S(x_S) f_V(x_v) dX = 0$ for $S \neq V$
     $\rightarrow$ any **two components do not share information**, meaning that, for example, the first order effect of feature X_1 and the interaction term of X_1 and X_2 are not correlated
  3. Variance Decomposition: Let $\sigma_f^2 = \int f(x)^2 dX$, then $\sigma^2(f) = \sum_{S \subseteq P} \sigma_S^2(f_S)$, where P is the set of all features
     $\rightarrow$ allows to **split the variance** of the function f among the components, and guarantees that it really adds up in the end

Appendix: Lagrange multipliers (LMs)

- **unconstrained optimization** problem
  - problem of finding the minimum/maximum of a function $f(x_1, x_2)$ subject to a constraint relating x_1 and x_2, written $g(x_1, x_2) = 0$
- LMs provide the extra degrees of freedom necessary to solve constrained optimization problems
- for f constrained by $g(x) = 0$: ∇f **and** ∇g **are parallel (or anti-parallel) vectors**, and so there must exist a parameter λ such that $\nabla f + \lambda \nabla g = 0$ where $\lambda \neq 0$ is known as a **Lagrange multiplier**
- **Lagrangian function**: $L(x, \lambda) = f(x) - \lambda g(x)$
  - critical points occur at **saddle points** rather than at local maxima (or minima) $\rightarrow$ traditional optimization techniques like gradient descent won't work

Appendix: Lagrange multipliers - "geometry"

- consider a D-dimensional variable x with components $x_1, \ldots, x_D$
- the constraint equation $g(x) = 0$ then represents a $(D-1)$-dimensional surface in x-space
- at any point on the **constraint** surface the gradient $\nabla g(x)$ of the constraint function will be **orthogonal to the surface**.
- we seek a point x^* on the constraint surface such that $f(x)$ is maximized
  - such a point must have the property that the vector $\nabla f(x)$ **is also orthogonal to the constraint surface**, because otherwise we could increase the value of $f(x)$ by moving a short distance along the constraint surface thus ∇f and ∇g are parallel (or anti-parallel) vectors

Appendix: Lagrange multipliers - "intuition" (1/2)

- if f draws circles, g a line, the critical point is where g is **tangent** to f
  - at the intersection with f, this is not critical point as f could be increased
  - if it does not intersect circles at all, there is no critical points
- e.g. to find the stationary point of
  - the function $f(x_1, x_2) = 1 - x_1^2 - x_2^2$
  - subject to the constraint $g(x1, x2) = x_1 + x_2 - 1 = 0$

Appendix: Lagrange multipliers - "intuition" (2/2)

- the corresponding Lagrangian function is hence:

$$L(x, \lambda) = 1 - x_1^2 - x_2^2 + \lambda(x_1 + x_2 - 1).$$

- critical point of L where $\nabla L = 0$
- the conditions for this Lagrangian to be stationary with respect to x_1, x_2 and λ give the following equations
  - $-2x_1 + \lambda = 0$
  - $-2x_2 + \lambda = 0$
  - $x_1 + x_2 - 1 = 0$

$\Rightarrow$ 3 variables, 3 equations $\rightarrow$ can be solved !

Appendix: Lagrange multipliers - final remarks

- the equation with respect to λ will always give the constraint function

- if we are only interested in critical point(s), then we can eliminate λ from the stationarity equations without needing to find its value (hence LM aka "**undetermined multiplier**")

- the LM gives the rate of change of the solution to the constrained maximization problem as the constraint varies

Appendix: BDMM

- use Lagrangian to solve constrained optimization in the context of neural networks
- **adaptation of the method of multipliers** introduced by Platt and Barr 1988
- use a single gradient descent to find both the optimal parameters and LM simultaneously
  - follow the gradient of the Lagrangian **downwards for the parameters** (gradient descent)
  - but **upwards for the LMs** λ (gradient ascent)

- *upsides*: **works well on convex case**; both losses are considered at every gradient step $\rightarrow$ applicable with stochastic GD as well
- *downside*: on concave Pareto case, it does not converge and keeps oscillating (bias of cherry-picking when to stop the optimization process !)

Appendix: MDMM

- modification of the BDMM with **more robust convergence properties**, also proposed by Platt and Barr 1988
- alter the BDMM to have a region of positive damping surrounding the constrained minima
- as BDMM is completely compatible with the penalty method, adds a penalty force corresponding to an quadratic energy

$$E_{penalty} = \frac{c}{2}(g(x))^2$$

- λ as the potential energy of an oscillating system
  - **introduce damping** on this energy to prevent the system from oscillating eternally and make it converge

Appendix: MDMM

- *upside*: **works well on convex and concave Pareto front** and works for stochastic GD
- *downside*: introduces an **additional damping hyper-parameter**, which:
  - trades the time to find the Pareto front with the time to converge to a solution on that front
  - but does not alter which solution is found, only how fast it is found

Some references

- https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/constrained-optimization/a/interpretation-of-lagrange-multipliers
- https://www.engraved.blog/how-we-can-make-machine-learning-algorithms-tunable
- https://christophm.github.io/interpretable-ml-book/
- Bishop 2007
- Platt and Barr 1988