Least Squares in Python

Manish Agarwal

Feb 2021

These notes are based on part 3 of the vmls book by Steven Boyd and Lieven Vandenberghe. We have worked out most of the in-chapter examples in the Python environment.

Contents

1 Least Squares

$\underset{(m\times n)}{A}\underset{(n\times 1)}{x} = \underset{(m\times 1)}{b}$ represent a system of m linear equations with n variables. If $m > n$ it is an over-determined system and under-determined if $m < n$. $Ax = 0$ is called a homogeneous set of equations with $x = 0$ as the trivial solution.

Suppose that the $m \times n$ matrix A is tall, so the system of linear equations $Ax = b$, where b is an m-vector, is over-determined. This system has a solution only if b is a linear combination of columns of A, which is in general not true. As a compromise, we seek an x for which $r = Ax - b$, the residual, is as small as possible. We thus, minimize the norm of the residual, $\|Ax - b\|$, which is equal to minimizing its square $\|r\|^2$, the sum of the squares of the residuals. The problem of finding an n-vector $\hat{x}$ that minimizes $\|Ax - b\|^2$, over all possible choices of x, is called the least squares problem, denoted by

$$\underset{x}{minimize}\ \|Ax - b\|^2.$$

This is also called *regression*. We are trying to find the linear combination of column vectors that is closest to the m-vector b, with the vector $\hat{x}$ giving us the coefficients.

If the columns of A are linearly independent then the solution involves the normal equations $A^TA\hat{x} = A^Tb$, where A^A is the Gram matrix associated with A; its entries are inner product of the columns of A, which is invertible. The unique final solution is given by

$$\hat{x} = (A^TA)^{-1}A^Tb = A^\dagger b.$$

$A^\dagger$ is the left inverse of A, which means that $\hat{x} = A^\dagger b$, showing that the solution $\hat{x}$ is a linear function of b.

In terms of the rows $\tilde{a}_i^T$ of the matrix A, we can write

$$\hat{x} = (A^A)^{-1}A^Tb = \left(\sum_{i=1}^{m} \tilde{a}_i\tilde{a}_i^T\right)^{-1}\left(\sum_{i=1}^{m} b_i\tilde{a}_i\right).$$

We express the $n \times n$ Gram matrix A^TA as a sum of m outer products, and the n-vector A^Tb as a sum of m different n-vectors. The optimal residual $\hat{r} = A\hat{x} - b$ is orthogonal to Az where z is any n-vector, i.e. $\hat{r}$ is orthogonal to all columns of A. This is called the *orthogonality principle*.

If $\underset{n\times k}{A} = \underset{(n\times k)}{Q}\underset{(k\times k)}{R}$ is the QR factorization (Gram-Schmidt algorithm) of A, with $Q^TQ = I$ and R the upper triangle matrix, we have the solution as

$$\hat{x} = R^{-1}Q^Tb$$

```
import numpy as np, scipy.linalg as scla
q, r = np.linalg.qr(a)
a_inv = scla.solve_triangular(r, q.T, check_finite=False, lower=False)
x_hat = a_inv @ b
```

The complexity of QR factorization is $\mathcal{O}(mn^2)$ and the back-substitution is $\mathcal{O}(n^2)$ giving a total of $\mathcal{O}(mn^2)$ run time. A direct inverse of A^TA would take $\mathcal{O}(n^3)$ steps. If A is sparse, sparse QR factorization can speed up the calculation quite a bit. We can also solve the normal equation $A^TA\hat{x} = A^Tb$ to take advantage of the sparsity in A

$$\begin{bmatrix} 0 & A^T \\ A & I \end{bmatrix}\begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} = \begin{bmatrix} 0 \\ b \end{bmatrix}.$$

To minimize $\|AX - B\|^2$ where X is a $n \times k$ matrix, A is a $m \times n$ matrix and B is a $m \times k$ matrix, we can still use the same setup. This is called the matrix least squares and is equivalent to solving set of k ordinary least square problems. This can be efficiently solved by taking the QR factorization of A once, and then solving for each of the k columns of B with a cost of $\mathcal{O}(mn^2 + mnk + kn^2)$, when k is small, this is quite efficient.

For a square of tall matrix A with independent columns, the left pseudo-inverse is given by $A^\dagger = (A^TA)^{-1}A^T$. For a square matrix it is same as A^{-1}. For a square of wide matrix A with independent rows the right pseudo-inverse is given by $A^\dagger = A^T(AA^T)^{-1}$. Again, for square matrix it is same as A^{-1}. Even when the columns or rows are not independent these formulas hold. $A = QR$ factorization can be used to calculate pseudo-inverse as well. Noting that $A^TA = R^TR$, we have the left inverse as $A^\dagger = R^{-1}Q^T$ and right inverse as $A^\dagger = QR^{-T}$. This gives us a way to solve over and under determined systems, respectively.

Example 1. *(Image reconstruction from line integrals) We explore the simple version of tomography problem here. We consider a square region, which we divide into $n \times n$ array of square pixels as shown in fig 1(a). The pixels are index column first, by a single index i ranging from 1 to n^2. We are interested in density which changes inside the region and for simplicity we assume constant density inside each pixel. We denote by x_i the density in pixel i, $i = 1, \ldots, n^2$. Thus, $x \in \boldsymbol{R}^{n^2}$ is a vector that describes the density across the rectangular array of pixels. The problem is to estimate the vector of densities x, from a set of sensor measurements that we*

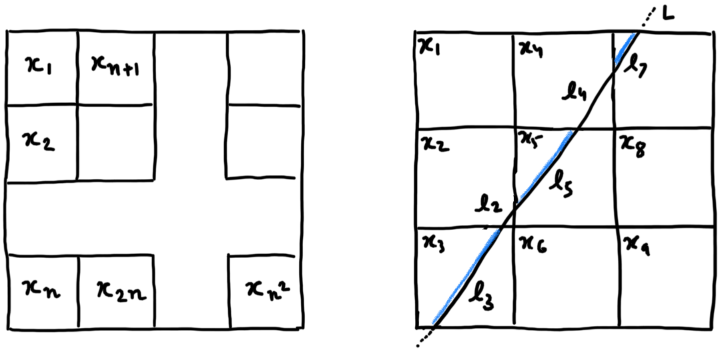

Figure 1: (a) The square region and its division into pixels. (b) An example of a 3 × 3 pixel patch, with a line L and its intersections l_i with the pixels. Clearly $l_1 = l_6 = l_8 = l_9 = 0$.

describe. Each sensor measurement is a line integral of the density over a line L. In addition, each measurement is corrupted by a small noise term. In other words, the sensor measurement for line L is given by $\sum_{i=1}^{n^2} l_i x_i + v$, *where* l_i *is the length of the intersection of line L with pixel i (or 0 if they don't intersect), and* v *is a small measurement noise. Figure 1(b) gives an example, with graphical explanation of* l_i. *Now suppose we have N line integral measurements, associated with lines* $L_1, \ldots, L_N$. *From these measurements, we want to estimate the vector of densities x. The lines are characterized by the intersection lengths* l_{ij}, $i = 1, \ldots, n^2$, $j = 1, \ldots, N$, *where* l_{ij} *gives the length of the intersection of line* L_j *with pixel i. Then, the whole set of measurements forms a vector* $y \in \boldsymbol{R}^N$ *whose elements are given by* $y_j = \sum_{i=1}^{n^2} l_{ij} x_i + v_j$, $j = 1, \ldots, N$. *We will reconstruct the pixel densities x from the line integrals measurements y given to us. A function 'line_pixel_length.py' is given to us that returns* $n \times n$ *matrix, whose* i, j*th element corresponds to the intersection lengths for a given line.*

Though irrelevant to the solution this is actually a simple version of tomography, best known for its application in medical imaging as the CAT scan. If an x-ray gets attenuated at rate x_i *in pixel i, a little piece of a cross-section in your body, the jth measurement is* $z_j = \prod_{i=1}^{n^2} e^{-x_i l_{ij}}$, *with the* l_{ij} *as before. Now define* $y_j = -log z_j$, *and we get the form* $y_j = \sum_{i=1}^{n^2} x_i l_{ij}$.

Solution: The attempt here is to restate the information in the form $y = Ax + v$. Here $y \in \boldsymbol{R}^N$ is the given measurement, $x \in \boldsymbol{R}^{n^2}$ is the physical quantity we are interested in, $A \in \boldsymbol{R}^{N \times n^2}$ is the relation between them, and $v \in \boldsymbol{R}^N$ is the noise, or measurement error. To find the elements of A we simply note that $A_{ji} = l_{ij}$, $j = 1, \ldots, N$, $i = 1, \ldots, n^2$. A turns out to be full rank so we can apply least squares to determine x.

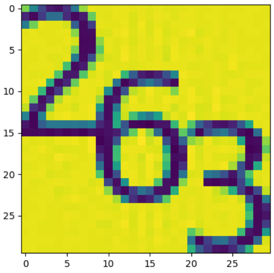

Figure 2: Reconstructed image from the least square solution.

```
from Boyd.tomodata import lines_d, lines_theta, y, N, n_pixels
L = np.zeros((N, n_pixels**2))

for j in range(N):
    L[j] = line_pixel_length(lines_d[j], lines_theta[j], n_pixels).flatten()

```

```
q,r = np.linalg.qr(L)
a_inv = np.linalg.solve(r, q.T)
x = a_inv @ y

x=x.reshape(n_pixels, n_pixels)
plt.imshow(x)
```

The output of the reconstructed image is shown in fig.2 □

2 Least squares data fitting

If we believe that an n-vector x and a scalar y are related by a function $f : \boldsymbol{R}^n \to \boldsymbol{R} : y \approx f(x)$, we can model then using a linear in parameter relationship as $y \approx \hat{f}(x) = \theta_1 f_a(x) + \ldots + \theta_p f_p(x)$, where $f_i : \boldsymbol{R}^n \to \boldsymbol{R}$ are basis functions or feature mappings and θ_i are the model parameters. Once the basis functions (to represent f) has been chosen, we need to choose model parameters. For data sample i, our model predicts the value $\hat{y}^{(i)} = \hat{f}(x^{(i)})$, so the prediction error or residual for this data point is $r^{(i)} = y^{(i)} - \hat{y}^{(i)}$. In vector notation, for the whole data d, we can write $r^d = y^d - \hat{y}^d$. *RMS prediction error* $rms(r^d)$ is a natural measure of how well the model predicts the observed data. The ratio $rms(r^d)/rms(y^d)$ gives a relative prediction error.

To choose the parameters $\theta_1, \ldots, \theta_p$ we minimize the RMS prediction error, i.e. $\|r^d\|^2$ giving rise to least square problem $y^d = A\theta$, where $A_{ij} = \hat{f}_j(x^{(i)})$, for $i = 1, \ldots, N$, $j = 1, \ldots, p$, with the solution $\hat{\theta} = A^\dagger y^d$. The number $\|y^d - A\hat{\theta}\|^2$ is called the minimum sum square error, the number $\frac{1}{N}\|y^d - A\hat{\theta}\|^2$ is called the minimum mean square error MMSE. Its square root is the minimum RMS fitting error.

- **Least square fit with a constant**. If we take $p = 1$ with $f_1(x) = 1$ for all x, least square fitting gives $\hat{\theta}_1 = (A^T A)^{-1} A^T y^d = avg(y^d)$, where we use $A = \mathbf{1}$, a $N \times 1$ matrix. The RMS fit to the data is $rms(y^d - avg(y^d)\mathbf{1}) = std(y^d)$, the standard deviation of the data.

- **Straight-line fit**. We take the basis functions $f_1(x) = 1$ and $f_2(x) = x$ and the model has the form $\hat{f}(x) = \theta_1 + \theta_2 x$, which is a straight line. Here $A = \begin{bmatrix} \mathbf{1} & x^d \end{bmatrix}$. The solution is $\hat{\theta}_2 = \frac{\sigma_{y^d}}{\sigma_{x^d}}\rho$ and $\hat{\theta}_1 = avg(y^d) - \hat{\theta}_2 avg(x^d)$, where ρ is the correlation between y^d and x^d. The final fitted equation has an intuitive form

$$\frac{\hat{y} - \overline{y}^d}{\sigma_{y^d}} = \rho\frac{x - \overline{x}^d}{\sigma_{x^d}}.$$

- **Trend and seasonal component**. Suppose the data represents a series of samples y at time $x^{(i)} = i$. The straight line fit to the time series $\hat{y}^{(i)} = \theta_1 + \theta_2 i$, $i = 1, \ldots, N$ is called the trend line. Subtracting the trend line from the original time series we get the de-trended time series, $y^d - \hat{y}^d$. Essentially, for the series $y^d = (y^{(1)}, \ldots, y^{(N)})$ of length N we represent it as $y^d \approx \hat{y}^d = \hat{y}^{const} + \hat{y}^{lin} = \theta_0 \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} + \theta_1 \begin{bmatrix} 1 \\ 2 \\ \vdots \\ N \end{bmatrix}$.
The de-trended series, many times, has seasonal component we can then have a seasonal term $y^d \approx \hat{y}^d = \hat{y}^{lin} + \hat{y}^{seas} = \theta_1 \begin{bmatrix} 1 \\ 2 \\ \vdots \\ N \end{bmatrix} + \begin{bmatrix} \theta_{2:(P+1)} \\ \theta_{2:(P+1)} \\ \vdots \\ \theta_{2:(P+1)} \end{bmatrix}$. The last term is the periodic or seasonal component with period P and consists of the pattern $(\theta_2, \ldots, \theta_{P+1})$ repeated N/P times. The constant term is redundant and removed as it has the same effect as adding a constant to the parameters $\theta_2, \ldots, \theta_{P+1}$. The least square fit is

computed by minimizing $\|A\theta - y^d\|^2$ where θ is a $(P+1)$-vector and the matrix A is given by

$$A = \begin{bmatrix} 1 & 1 & 0 & \dots & 0 \\ 2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ P & 0 & 0 & \dots & 1 \\ P+1 & 1 & 0 & \dots & 0 \\ P+2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 2P & 0 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & & \vdots \\ N-P+1 & 1 & 0 & \dots & 0 \\ N-P+2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ N & 0 & 0 & \dots & 1 \end{bmatrix}$$

The residual or prediction error in this case is called the de-trended, seasonally-adjusted series.

- **Polynomial fit**. A simple extension beyond the straight-line fit is the polynomial fit with $f_i(x) = x^{i-1}$, $i = 1, \dots, p$, so $\hat{f}$ is a polynomial of degree at most $p-1$, $\hat{f}(x) = \theta_1 + \theta_2 x + \dots + \theta_p x^{p-1}$. The A matrix in this case is called the **Vandermonde matrix**

$$A = \begin{bmatrix} 1 & x^{(1)} & \dots & (x^{(1)})^{p-1} \\ 1 & x^{(2)} & \dots & (x^{(2)})^{p-1} \\ \vdots & \vdots & & \vdots \\ 1 & x^{(N)} & \dots & (x^{(N)})^{p-1} \end{bmatrix}.$$

- **Piecewise-linear fit**. A piecewise linear function with knot/kink points $a_1 < a_2 < \dots < a_k$ is a continuous function that is affine in between the knot points. Any piecewise linear function with k knot points can be described using $p = k+2$ basis functions $f_1(x) = 1$, $f_2(x) = x$, $f_{i+2}(x) = (x - a_i)_+$, $i = 1, \dots, k$, where $(u)_+ = max\{u, 0\}$.

- **Regression**. For the general case where x is a n-vector we have $haty = x^T\beta + v$, where β is the weight vector and v is the offset. By using the basis $f_1(x) = 1$ and $f_i(x) = x_{i-1}$, $i = 2, \dots, n+1$ we can write $\hat{y} = \theta_1 + x^T\theta_{2:n+1}$. The $N \times (n+1)$ matrix A is given by $\begin{bmatrix} \mathbf{1} & X^T \end{bmatrix}$, where X is the feature matrix with columns $x^{(1)}, \dots, x^{(N)}$. The regression model, thus, is a special case of our general data fitting model. Conversely, we can think of regression as out linear in parameter model with different set of feature vectors of dimension $p-1$.

- **Auto-regression time series model**. For a time series $z_1, z_2, \dots$ the AR model with M lag or memory is given by $\hat{z}_{t+1} = \theta_1 z_t + \dots + \theta_M z_{t-M+1}$, $t = M, M+1, \dots$. This can be estimated using least squares as well to find the AR parameters by minimizing the prediction error $z_t - \hat{z}_t$ over $t = M+1, \dots, T$.

Example 2. *We are given the time series of hourly temperature at Chicago Airport for June 2016 with 688 observations, see fig.3. Fit a constant prediction, one-step-back predictor, 24-hour-ago predictor, and AR model with 8 lags model. Compare the RMS of the models.*

Solution: A simple constant prediction model with $\hat{z}_{t+1} = 21.6^oC$ (the average temperature) has RMS prediction error 4.44^oC (standard deviation). The very simple predictor $\hat{z}_{t+1} = z_t$, guessing that the temperature next hour is the same as the current temperature, has RMS error or 1.99^oC. The predictor $\hat{z}_{t+1} = z_{t-23}$, i.e. guessing that the temperature next hour is what it was yesterday at the same time, has RMS error 5.69^oC. We then fit a AR(8) model with 680 observations to get and RMS of 1.91^oC.

```
# constant model
pred = pd.Series(a.mean(), index=a.index)
rms = np.sqrt(((a-pred)**2).mean())
```

```
# one-step-back
pred = a.shift(1)
rms = np.sqrt(((a-pred)**2).mean())

# 24-hr-back
pred = a.shift(24)
rms = np.sqrt(((a-pred)**2).mean())

# AR(8) model
y = a.values[8:][:,None]
x = np.ones((680, 9))
for i in range(1,9):
    x[:,i] = a.shift(i).values[8:]

q, r = np.linalg.qr(x)
a_inv = np.linalg.solve(r, q.T)
b = a_inv @ y

y_hat = x @ b
rms = np.sqrt(np.mean((y-y_hat)**2))
```

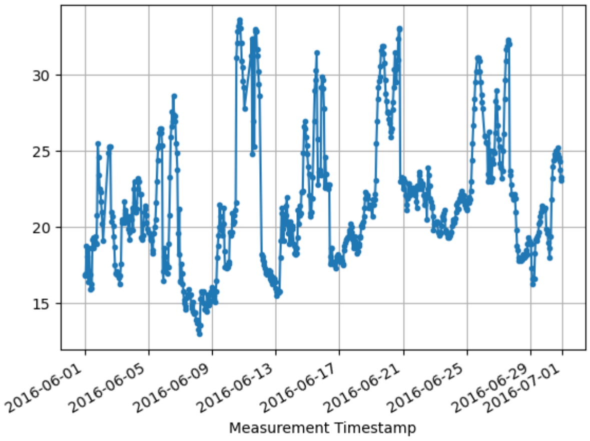

Figure 3: Hourly temperature at Chicago airport.

□

When the dependent variable y is positive and varies over a large range, it is common to replace it with its logarithm $w = \log y$, and then use least squares to develop a model for w, $\hat{w} = \hat{g}(x)$. We then form our estimate y using $\hat{y} = e^{\hat{g}(x)}$. When we fit a model $\hat{w} = \hat{g}(x)$ to the logarithm $w = \log y$, the fitting error for w can be interpreted in terms of the percentage or relative error between $\hat{y}$ and y, $\eta = max\{\hat{y}/y, y/\hat{y}\} - 1$. The connection to the relative error between $\hat{y}$ and y and the residual r in predicting w is $\eta = e^{|r|} - 1$. For example, a residual with $|r| = 0.05$ corresponds to a relative error in our prediction $\hat{y}$ or y of 5%, using $e^{|r|} - 1 \approx |r|$ for small r.

2.1 Validation

The aim of model fitting is to achieve a good fit on new data that we have not yet seen. One very common assumption is that the data are described by a formal probability model, which can then be used to make predictions and understand variance of it. Good generalization is the primary aim, but overfitting is what generally ends up happening in practice. out-of-sample validation with 80-20 or 90-10 random divide is commonly used. We fit the model on the training set and then calculate the RMS on the test set. If the RMS is similar on the two sets we have confidence in the generalizability of the data. But there is no guarantee of this, without the assumption that the future data will look like the test data. A more confident approach is k-fold cross-validation. When the RMS prediction error on the training set is much smaller than the RMS prediction error on the test set, we say that the model is over-fit. One method to reduce over-fitting is to keep the model simple or regularize it. In general, we seek a model that makes good predictions on the training set and also makes good predictions on the test data set - i.e. we seek good performance and generalization ability simultaneously. To choose among the various model we should choose a model that has test set RMS error that is near the minimum over the candidates, and then choose the simplest one among these similar error sub-candidates.

Cross-validation can give us more confidence and can also be used to check for the stability of the model coefficients. The RMS cross-validation error for k folds is $\sqrt{\frac{1}{k}\sum_{i=1}^{k}\varepsilon_i^2}$. Cross-validation does not check a particular model but check a selection of basis functions. One can either fit the model over all the data with the chosen basis or use the average of the model parameters from the different folds. In time series models, to avoid look-ahead, the training set for prediction model is typically taken to be the data examples up to some point in time, and the test data are chosen as points that are past that time.

2.2 Feature engineering

Choosing the feature mapping functions is called feature engineering, since we are generating features to use in regression. In many cases the basis functions include the constant one, $f_1(x) = 1$ as well as the original data set as features $f)i(x) = x_{i-1}$, $i = 2, \ldots, n+1$. Adding features efficacy can be checked using cross-validation. In case of too many original features, we can use dimension reduction or data aggregation to reduce the features. Transforming features is a very common action which includes:

- standardizing - $f_i(x) = (x_i - b_i)/a_i$, $i = 2, \ldots, n+1$, so that across the data the mean is near 0 and standard deviation is around 1. This is also called z-scoring.

- Winsorizing - It is common to clip the data to remove errors and large values.

- log transform - with positive features that vary over wide range, it is common to take their logarithms. If the data has 0 as well, $log(x_k + 1)$ is commonly used. This compresses the range of values we encounter.

It is also very common to create new features:

- Expanding categoricals: expanding a categorical feature with l values means replacing it with a set of $l-1$ new features, each of which is Boolean, and simply records whether or not the original feature has the associated value. The original feature has the default value. This is also called *one-hot-encoding.*

- Generalized additive model: A feature like $\min\{x_i + a, 0\}$ is the amount by with feature x_i is below $-a$. This when generalized lead to the predictor $\hat{y} = \psi_1(x_1) + \ldots + \psi_n(x_n)$, where ψ_i is the piecewise-linear function $\psi_i(x_i) = \theta_{n+1}\min\{x_i + a, 0\} + \theta_i x_i + \theta_{2n+i}\max\{x_i - b, 0\}$, which has kink or knot points at the values $-a$ and $+b$. The model has $3n$ parameters, corresponding to the original features, and two additional features per original feature. This is called the *generalized additive model.* This allows for different slopes across the knot points.

- Products and interactions: Pairs of original features, as products can introduce interactions. Boolean features in a product are easier to interpret as well.

- Stratified models - We an carry out clustering of the original feature vectors, and fit a separate model within each cluster. To evaluate $\hat{y}$ for a new x, we first determine which cluster x is in, and then use the associated model. Out-of-sample validation can be used to check if stratified models work.

Some advanced feature generation methods are:

- custom mappings - we might use highest and lowest values over the last week or some period. Word count can be replaced by term frequency inverse document frequency to give high weight to uncommon words

- predictions from other models - Predictions of other models are, often, used as features along with raw data.

- distance to cluster representatives - Clustering the data into k groups we can have distance from these clusters as a new feature $f(x) = e^{-\|x-z_i\|^2/\sigma^2}$ where σ is a parameter.

- random features - nonlinear function of a random linear combination of original features can also serve as a new feature. To add K new features we first generate a random $K \times n$ matrix R. We then generate new features as $(Rx)_+$ or $|Rx|$ where $(.)_+$ and $|.|$ are applied element wise to the vector Rx.

- neural network features - NN can find good feature mappings directly from the data, provided there is a large amount of data available.

Thus this is more of an art. We should try simple models first, starting from constant, then a simple regression model, and compare more sophisticated models against these. Adding new features will always improve training error, but the important question is whether or not it substantially reduces the RMS error on the test or validation set. A small reduction in test set error is not meaningful. Finally, adding new features can easily lead to over-fit. Thus using simpler models or regularization should go hand in hand with feature engineering.

Example 3. We fit a regression model for the house sales data here showing the importance of feature engineering. For the simple model we simply use the 4 raw features and get the coefficients via least squares. For the complex model we create a generalized additive features along with some dummy variables. The improvement is noticable.

```
from sklearn.preprocessing import OneHotEncoder

data = house_sales_data()
data = pd.DataFrame(data)
y = data.price

def ls(y, A):
    q, r = np.linalg.qr(A)
    a_inv = scipy.linalg.solve_triangular(r, q.T, check_finite=False, lower=False)
    x = a_inv @ y
    return pd.Series(x, index=A.columns)

# simple model
A = data[['area', 'beds', 'condo', 'location']]
beta_simple = ls(y, A)
area        145.495201
beds        -12.116593
condo       -16.543274
location     14.249942

# complex model
B = pd.DataFrame({'one': 1}, index=data.index)
B['area'] = data.area
B['ExArea'] = data.area.sub(1.5).clip(lower=0)
B['beds'] = data.beds
B['condo'] = data.condo
encodes = pd.DataFrame(OneHotEncoder().fit(data[['location']]).
                       transform(data[['location']]).toarray(),
                       columns=range(1,5), index=data.index).astype(int)
B = pd.concat([B, encodes.iloc[:,1:]], axis=1)
beta_complex = ls(y, B)
one       115.616824
area      175.413141
ExArea    -42.747768
beds      -17.878355
condo     -19.044726
2        -100.910503
3        -108.791122
4         -24.765247

from sklearn.metrics import mean_squared_error as MSE, \
    mean_absolute_error as MAE, \
    mean_absolute_percentage_error as MAPE

pd.DataFrame({'simple': [np.sqrt(MSE(y, A@beta_simple)),
                                 MAE(y, A@beta_simple), MAPE(y, A@beta_simple)],
              'complex':[np.sqrt(MSE(y, B@beta_complex)),
                                    MAE(y, B@beta_complex), MAPE(y, B@beta_complex)]},
             index = ['RSE', 'MAE', 'MAPE'])
         simple    complex
RSE   75.036413  68.344287
MAE   55.044768  50.713427
MAPE   0.273905   0.259596
```

To validate the model, we use 5-fold cross-validation. The training and test errors are similar, so our model is not an overfit and the improvement with complex features is stable. The table also shows the coefficients are reasonably stable across the different folds, giving us more confidence in the model.

```
from sklearn.metrics import mean_squared_error as MSE, \
    mean_absolute_error as MAE, \
    mean_absolute_percentage_error as MAPE
pd.DataFrame({'simple': [np.sqrt(MSE(y, A@beta_simple)),
                                   MAE(y, A@beta_simple), MAPE(y, A@beta_simple)],
              'complex':[np.sqrt(MSE(y, B@beta_complex)),
                                   MAE(y, B@beta_complex), MAPE(y, B@beta_complex)]},
             index = ['RSE', 'MAE', 'MAPE'])

from sklearn.model_selection import cross_validate, cross_val_score
from sklearn.linear_model import LinearRegression as LS
from sklearn.metrics import make_scorer
scoring = {'MSE': make_scorer(MSE, greater_is_better=False),
           'MAE': make_scorer(MAE, greater_is_better=False),
           'MAPE': make_scorer(MAPE, greater_is_better=False)}
scores = cross_validate(LS(), B, y, scoring=scoring, cv=5,
                    return_train_score=True, return_estimator=True)
-pd.DataFrame(scores).iloc[:,3:].round(2)
"""
   test_MSE  train_MSE  test_MAE  train_MAE  test_MAPE  train_MAPE
0   4217.95    4793.90     48.89      51.47       0.27        0.26
1   5094.19    4590.65     48.03      51.35       0.25        0.26
2   7009.15    4128.08     64.35      47.04       0.29        0.25
3   2582.74    5214.24     41.71      53.48       0.23        0.27
4   5237.62    4542.28     53.72      50.07       0.28        0.26
"""
coef = pd.DataFrame(0, index=B.columns, columns=range(1,6))
for i, item in enumerate(scores['estimator']):
    coef[i+1] = item.coef_
    coef[i+1]['one'] = item.intercept_
coef.T.round(2)
"""
      one    area  ExArea   beds  condo       2       3      4
1  119.94  169.32  -35.71 -14.99 -18.39 -104.39 -115.17 -29.05
2  105.29  175.79  -44.30 -20.77 -18.16  -83.78  -88.11  -4.99
3  136.77  178.56  -41.71 -21.52 -25.48 -116.11 -126.75 -52.26
4  106.20  186.83  -53.70 -18.51 -19.50 -102.44 -109.37 -22.40
5  109.22  166.61  -38.55 -13.69 -14.49  -96.79 -103.26 -14.98
"""
```

□

3 Least squares classification

In a classification problem the aim is to predict categories. For a binary case, for a given data point x, y, with total N data points, with predicted output $\hat{y} = \hat{f}(x)$, there are only four possibilities - true positive N_{tp}, true negative N_{tn}, false positive N_{fp}, false negative N_{fn}. The third case is called **type I** error and the fourth case is called **type II** error. In some applications we care equally about making the two types of errors; in others we may care more about making one type of error than other. Counting each of the four possibilities in a table is called a *confusion matrix*. Various performance metrics are expressed in terms of the numbers in the confusion matrix - error rate $\frac{N_{fp}+N_{fn}}{N}$, true positive rate/sensitivity/recall rate $\frac{N_{tp}}{N_{tp}+N_{fn}}$, false positive rate/false alarm rate $\frac{N_{fp}}{N_{fp}+N_{tn}}$, the specificity or true negative rate = 1 - false positive rate $\frac{N_{tn}}{N_{fp}+N_{tn}}$, precision $\frac{N_{tp}}{N_{tp}+N_{fp}}$. A good classifier will have small error rate and false positive rate, and high true positive rate, true negative rate and precision. We may care more about one or these matrices than the other, based on the application.

Logistic regression and **support vector machines** are much more superior in terms of performance than least squares classifier which essentially uses $\hat{f}(x) = sign(\tilde{f}(x)))$, where $\tilde{f}(x) = x^T\beta$ is the prediction of the linear model. The number $\tilde{f}(x)$ can be related to our confidence in our guess $\hat{y} = sign(\tilde{f}(x))$.

Example 4. We now tackle a much larger problem, the MINST data set of image classification. We will compare the least square and logistic classifiers and also look at some feature engineering. Each 60000 of the images is 28×28 pixels of which we remove the ones on the boundary for with the nonzero value is less than for 600 training examples. The remaining 493 pixels are used as the features. We also have 10000 remaining images as the validation set on which we validate our results. We use an additional feature with value 1, as the $n = 494$ features in the least square classifier.

```
# MINST TRAIN DATA
from keras.datasets import mnist
data = mnist.load_data()
y = data[0][1]
x = data[0][0].reshape(60000, 28*28)/255
print(y.shape, x.shape)
>> (60000,) (60000, 784)
pd.Series(y).value_counts()

# border removal data data
mask = np.sum(x!=0, axis=0)>=600
xx=x[:,mask]
print(xx.shape)
>> (60000, 493)

from sklearn.linear_model import LinearRegression, LogisticRegression

# linear classifiation
ls = LinearRegression(fit_intercept=True).fit(xx,y==0)
ls_beta = np.zeros(28*28)
ls_beta[mask] = ls.coef_

g=sb.heatmap(-ls_beta.reshape(28,28), mask=~mask.reshape(28, 28),
                cmap='RdBu', center=0, vmin=-0.1, vmax=0.1)
g.set_facecolor('xkcd:black')

yh = ls.predict(xx)
from sklearn.metrics import confusion_matrix
confusion_matrix(y==0, yh>0.5)
# array([[53910,   167],
       # [  765,  5158]])

def metric(truth, prediction):
    return pd.Series({'error rate':np.sum(truth!=prediction)/len(truth),
            'true positive rate': np.sum(truth & prediction)/np.sum(truth),
            'false positive rate': np.sum(~truth & prediction)/np.sum(~truth)}).round(4)*100
metric(y==0, yh>0.5)
>>>
error rate              1.55
true positive rate     87.08
false positive rate     0.31

ax = pd.Series(yh[y==0]).hist(bins=100, color='blue', density=True, alpha=0.6)
pd.Series(yh[y!=0]).hist(ax=ax, bins=100, color='red', density=True, alpha=0.6)
ax.axvline(x=0.5, color='k')

# validation
```

```
y_val = data[1][1]
x_val = data[1][0].reshape(10000, 28*28)/255
y_val_hat = ls.predict(x_val[:,mask])
confusion_matrix(y_val==0, y_val_hat>0.5)
# array([[8978,   42],
#        [ 116,  864]])
metric(y_val==0, y_val_hat>0.5)
>>>
error rate               1.58
true positive rate      88.16
false positive rate      0.47
```

The performance on the training data set is shown in the confusion matrix. The error rate is $\frac{167+765}{60000} = 1.55\%$, the true positive rate is $\frac{5158}{5158+765} = 87.08\%$, and the false positive rate is $\frac{167}{167+53910} = 0.31\%$. We validate the model coefficients on the validation test and obtain the confusion matrix. for the test set the error rate is $\frac{42+116}{10000} = 1.58\%$, the true positive rate is $\frac{864}{864+116} = 88.16\%$, and the false positive rate of $\frac{42}{42+8978} = 0.47\%$. These are similar to the test set, which suggests that our classifier is not over-fitting. Figure 4 shows the coefficients and the histogram of the linear prediction for the two classes.

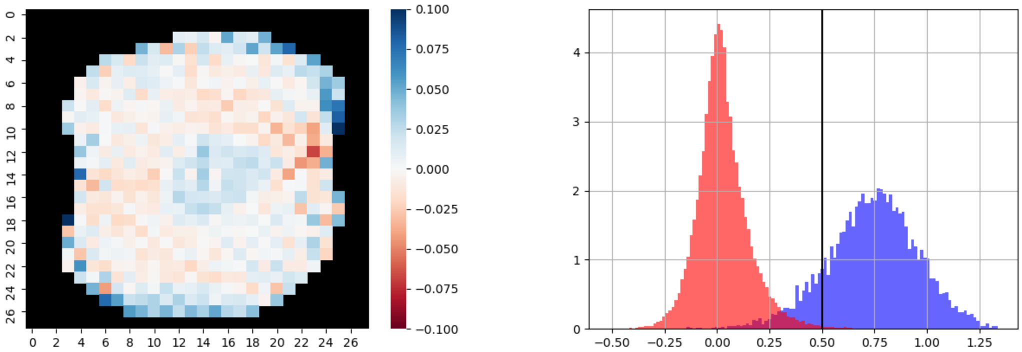

Figure 4: The coefficient β_k in the least squares classifier that distinguishes the digit zero from the other nine digits. The distribution of $\tilde{f}(x)$ in the boolean classifier for recognizing the digit zero, over all elements of the training set. The red bars correspond to the digits 1-9, the blue bars correspond to the digit 0. The black line shows the decision boundary.

We now try the Logistic regression and calculate the confusion matrix for train and validation and get an error rate of 0.6% for training and 0.8% for validation set. The true positive rate is 96.39% for training and 96.63% for validation set. Finally, the false positive rate is 0.27% for training and 0.52% for validation set. Clearly the Logistic regression model is superior as pointed out before.

```
lr = LogisticRegression(fit_intercept=True).fit(xx, y==0)
lr_hat = lr.predict(xx)
metric(y==0, lr_hat)
error rate               0.60
true positive rate      96.39
false positive rate      0.27
metric(y_val==0, lr.predict(x_val[:,mask]))
error rate               0.80
true positive rate      96.63
false positive rate      0.52
```

We now apply simple random feature engineering with the aim to improve our classifier. We add 5000 new features to the original 494 features. We first generate a 5000 × 494 matrix R with random chosen entries ±1. The 5000 new functions are then given by $max\{0, (Rx)_j\}$, for $j = 1, \ldots, 5000$. With a total of 5494 features we refit the linear classification model and calculate the confusion matrix.

```
# feature engineering
R = np.random.choice([-1,1], (5000,494))
```

```
xt = np.concatenate([np.ones((60000,1)), xx],axis=1)
f = np.clip(xt @ R.T, 0, np.inf)
xf = np.concatenate([xt, f], axis=1)

ls = LinearRegression(fit_intercept=False).fit(xf,y==0)
metric(y==0, ls.predict(xf)>0.5)
# TRAIN SET ===
# error rate              0.21
# true positive rate     98.14
# false positive rate     0.03
x_val_t = np.concatenate([np.ones((10000,1)), x_val[:,mask]], axis=1)
val_f = np.clip(x_val_t @ R.T, 0, np.inf)
x_val_f = np.concatenate([x_val_t, val_f], axis=1)
metric(y_val==0, ls.predict(x_val_f)>0.5)
# VALIDATION SET ===
# error rate              0.25
# true positive rate     98.37
# false positive rate     0.10

lr = LogisticRegression(fit_intercept=False).fit(xf, y==0)
metric(y==0, lr.predict(xf))
# TRAIN SET ===
# error rate               0.0
# true positive rate     100.0
# false positive rate      0.0
metric(y_val==0, lr.predict(x_val_f))
# VALIDATION SET ===
# error rate              0.23
# true positive rate     98.88
# false positive rate     0.13
```

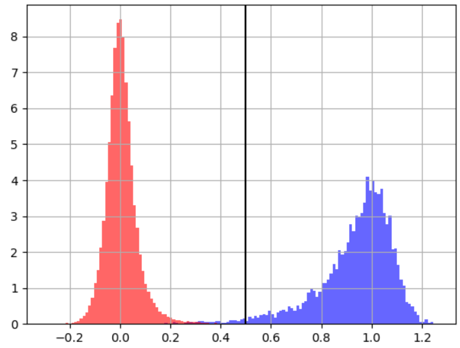

Figure 5: The effect of random features on the distribution of $\tilde{f}(x)$ in the boolean classifier for recognizing the digit zero, over all elements of the training set. The red bars correspond to the digits 1-9, the blue bars correspond to the digit 0. The black line shows the decision boundary.

These are some mind blowing results! The validation error is down to 1.63%, rivaling an average human performance! We added some random features and some non-linearity and linear classifier is not only as good as a logistic regression but it does not overfit, as the train and validation errors are close. Logistic regression however shows overfitting but its validation performance is still better than linear but only slightly so. The separation is neat in fig.5. This is black magic! This is similar to the kind of features a neural network works with and some correspondence can be drawn between the two. a value of 10000 random features improve the numbers only slightly. □

One useful modification of the least squares classifier is to skew the decision boundary, by subtracting a constant α from $\tilde{f}(x)$ before taking the sign $\hat{f}(x) = sign(\tilde{f}(x) - \alpha)$. We call α the decision threshold for the modified classifier. By choosing α we can play with the different errors related to the confusion matrix. Choosing a higher α decreases the true positive rate (which is bad), but it also decreases the false positive rate (which is good). The parameter is chosen depending on how much we care about the two competing metrics, in the particular application. By sweeping α over a range, we obtain a family of classifiers that vary in their true positive and false positive rates. A common way to plot this data is to use a *receiver operating characteristic* ROC curve. The ROC shows the true positive rate on the vertical axis and the false positive rate on the horizontal axis.

Example 5. We examine the skewed threshold least square classifier for the example above, where we attempt to detect whether or not a handwritten digit is zero. Figure 6 shows the error rates change with the α value. The same information can be shown on the ROC curve.

```
# ROC curve
ls = LinearRegression(fit_intercept=True).fit(xx,y==0)
yh = ls.predict(xx)
err = pd.DataFrame({alpha: metric(y==0, yh>alpha) for alpha in np.arange(-2,2,0.01)})
fig = plt.figure()
ax1 = fig.add_subplot(121)
err.T.plot(grid=True, ax=ax1)
ax2 = fig.add_subplot(122)
plt.plot(err.T['false positive rate'], err.T['true positive rate'])
ax2.grid(True)
ax2.set_title('ROC curve')
plt.tight_layout()
```

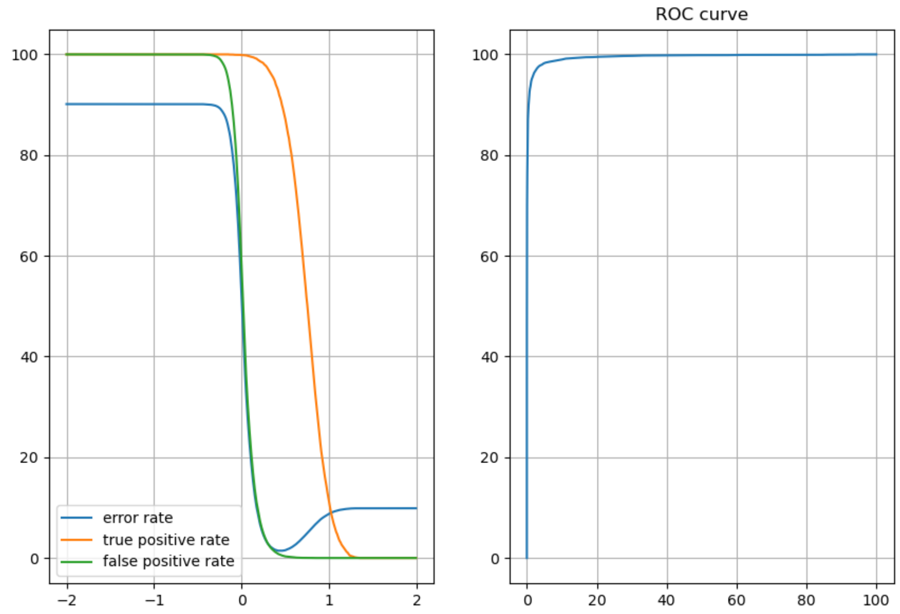

Figure 6: True positive, false positive, and total error rate versus decision threshold α. The ROC curve.

□

In a K-class classification problem we will encode the labels as $y = 1, 2, \ldots, K$. In some applications there are more natural encodings; for example, the Likert scale labels Strongly disagree, Disagree, Neutral, Agree, Strongly Agree are typically encoded as -2,-1,0,1,2 respectively. A multi-class classifier is a function $\hat{f} : \boldsymbol{R}^n \to \{1, \ldots, K\}$. The confustion matrix is now a $K \times K$ matrix with N_{ij} being the number of points for which $y = i$ and $\hat{y} = j$. We can then derive various measures of the accuracy of the predictions. We let N_i denote the total number of data points for which $y = i$, $N_i = \sum_{j=1}^{K} N_{ij}$. We have $N = \sum_{i=1}^{K} N_i$. The error rate is given by $\frac{1}{N} \sum_{i \neq j} N_{ij}$. This measure assumes that all errors are equally bad. The quantity $\frac{N_{ii}}{N_i}$ is called true label rate.

For each possible label value, we construct a new data set with the Boolean label $+1$ if the label has the given value and -1 otherwise. This is called one-versus-others classifier. Our final classifier is $\hat{f}(x) = arg \max_{1 \leq k \leq K} \tilde{f}_k(x)$, where $\tilde{f}_k$ is the least squares regression model for label k against the others. These can also be used to derive the rank of predicted class. We can also skew the decision threshold to trade off the true positive and false positive rates by using $\hat{f}(x) = arg \max_{1 \leq k \leq K} \left(\tilde{f}_k(x) - \alpha_k\right)$, where α_k are constants chosen to trade off the true label k rates.

If these K least square problems are solved independent the run time would be $\mathcal{O}(2KNp^2)$ where p is the number of features. But we can carry out the QR factorization just once and solve it simultaneously via matrix least squares. When K is small compared to p, that cost is about the same as solving just one least sqaures problem. Further, only first $K-1$ problems need to be solved. The y vectors when summed will be $(2-K)\mathbf{1}$. And since the least squares coefficients are linear combination of ys, we have $\hat{\theta}_1 + \ldots + \hat{\theta}_K = (2-K)e_1$, where $\hat{\theta}_k$ is the coefficient vector for distinguishing class k from the others, and e_1 is the least squares approximate solution when the y is $\mathbf{1}$. Assuming the first basis function is $f_1(x) = 1$, we have e_1 as the canonical basis vector with 1 at first position and all zeros elsewhere.

Example 6. We now use 10-class classification on the MINST dataset. We encode the digits $0, \ldots, 9$ with $k = 1, \ldots, 10$ and compute $\hat{f}_k = sign(x^T\beta_k + v_k)$, to distinguish digit k from others. The ten Boolean classifiers are combined into a multi-class classifier $\hat{f}(x) = arg \max_{1\leq k\leq 10}(x^T\beta_k + v_k)$.

```
# multi-class classification
from sklearn.preprocessing import OneHotEncoder
ohe = 2* OneHotEncoder().fit(y.reshape(-1,1)).transform(y.reshape(-1,1)).toarray() -1
ls = LinearRegression(fit_intercept=True).fit(xx,ohe)
def mcmetric(truth, prediction):
    sr = pd.Series({'error rate':np.sum(truth!=prediction)/len(truth)})
    for i in np.unique(truth):
        imask = truth == i
        sr['true %s rate'%(i)] = np.sum(prediction[imask]==i)/np.sum(imask)
    return sr.round(4)*100
er = {}
er['train']=mcmetric(y, np.argmax(ls.predict(xx),1))
er['test']=mcmetric(y_val, np.argmax(ls.predict(x_val[:,mask]),1))
ls = LinearRegression(fit_intercept=True).fit(xf,ohe)
er['train+f5K']=mcmetric(y, np.argmax(ls.predict(xf),1))
er['test+f5K']=mcmetric(y_val, np.argmax(ls.predict(x_val_f),1))
pd.DataFrame(er)
             train+f5K  test+f5K  train   test
error rate        0.59      2.26  14.45  13.93
true 0 rate      99.81     99.08  95.71  96.33
true 1 rate      99.69     99.21  97.05  97.53
true 2 rate      99.48     97.38  79.84  78.97
true 3 rate      99.12     97.43  84.00  87.52
true 4 rate      99.37     97.15  88.82  89.92
true 5 rate      99.43     97.65  73.31  73.54
true 6 rate      99.66     98.33  92.06  91.44
true 7 rate      99.20     96.69  86.88  85.89
true 8 rate      99.37     97.54  75.49  77.62
true 9 rate      98.92     96.83  79.71  79.58
```

The error rate on the test set is around 14%. Applying 5000 random features trick again, we see that the random 5000 features has worked it magic again reducing the test error to 2.3%. □

4 Multi-objective least squares

In some applications we have multiple objectives, all of which we would like to be small $J_1 = \|A_1x - b_1\|^2, \ldots, J_k = \|A_kx - b_k\|^2$. Here A_i is an $m_i \times n$ matrix, and b_i is the m_i-vector. We seek a single $\hat{x}$ that gives a compromise, and makes them all small, to the extent possible. We call this the multi-objective least squares problem, and refer to J_i, $0 \leq i \leq k$ as the k objectives. This is usually solved by minimizing a weighted sum objective

$$J = \sum_{i=1}^{k} \lambda_i J_i = \sum_{i=1}^{k} \lambda_i \|A_i x - b_i\|^2,$$

where $\lambda_1, \ldots, \lambda_k$ are positive weights, that express the relative desire for the terms to be small. We will discuss how to choose these weights later. It is common to assume $\lambda_1 = 1$ for the primary objective. We can minimize the weighted sum objective function by expressing it as a standard least square problem $J = \left\| \begin{bmatrix} \sqrt{\lambda_1}(A_1x - b_1) \\ \vdots \\ \sqrt{\lambda_k}(A_kx - b_k) \end{bmatrix} \right\|^2$, where we use the property $\|(a_1, \ldots, a_k)\|^2 = \|a_1\|^2 + \ldots + \|a_k\|^2$ for any vectors $a_1, \ldots, a_k$. So we have

$$J = \left\| \begin{bmatrix} \sqrt{\lambda_1} \underset{m_1 \times n}{A_1} \\ \vdots \\ \sqrt{\lambda_k} \underset{m_k \times n}{A_k} \end{bmatrix} x - \begin{bmatrix} \sqrt{\lambda_1} \underset{m_1 \times 1}{b_1} \\ \vdots \\ \sqrt{\lambda_k} \underset{m_k \times 1}{b_k} \end{bmatrix} \right\|^2 = \| \underset{m \times n}{\tilde{A}} x - \underset{m \times 1}{\tilde{b}} \|^2,$$

where $m = m_1 + \ldots + m_k$. Thus the problem is reduced to a standard least squares problem. Provided the columns of $\tilde{A}$ are linearly independent, the minimizer is unique and given by $\hat{x} = (\tilde{A}^T\tilde{A})^{-1}\tilde{A}^T\tilde{b} = (\lambda_1 A_1^T A_1 + \ldots + \lambda_k A_k^T A_k)^{-1}(\lambda_1 A_1^T b_1 + \ldots + \lambda_k A_k^T b_k)$. We can compute $\hat{x}$ via the QR factorization of $\tilde{A}$. The assumption that the columns of $\tilde{A}$ are linearly independent is not the same as assuming that each of $A_1, \ldots, A_k$ has linearly independent columns. If even just one of the matrices $A_1, \ldots, A_k$ has linearly independent columns, then A does. If $m_i < n$ for all i, i.e. A_i are wide and $m_1 + \ldots + m_k \geq n$, making $\tilde{A}$ tall or square with linearly independent columns, we can have the case were $\tilde{A}$ has linearly independent columns even when none of the matrices $A_1, \ldots, A_k$ do. For a bi-criterion problem a *Pareto optimal tradeoff curve* can be constructed by varying λ from 0 to ∞. With $k > 2$ we get an optimal trade-off surface. The secondary objectives (sometimes called regularization) generally correspond to the desire that some parameters be 'small' or 'smooth', or close to some previous value, corresponding to some previous knowledge. In data fitting we can use validation to help guide us in the choice of the weights. In exploring λ parameters we generally logarithmically space them as $\lambda^{min}, \theta\lambda^{min}, \theta^2\lambda^{min}, \ldots, \theta^{N-1}\lambda^{min} = \lambda^{max}$.

For $\|A\hat{x} - y\|^2$ as our primary objective, our prior information about x enters in one or more secondary objectives. $\|x\|^2$ would want x to be small, $\|x - x^{prior}\|^2$ would want x should be near x^{prior}, $\|Dx\|^2$ would want x to be smooth for D being the first difference matrix, and Dirichlet energy $\mathcal{D}(x) = \|A^T x\|^2$ would want x to be smooth across the graph (Laplacian regularization). For the objection $\|Ax - y\|^2 + \lambda\|x\|^2$ for some $\lambda > 0$, called Tikhonov regularization can be solved with the stacked matrix $\tilde{A} = \begin{bmatrix} A \\ \sqrt{\lambda} I \end{bmatrix}$, which always has linearly independent columns, without any assumption about A. The Gram matrix associated with $\tilde{A}$ is $A^TA + \lambda I$, which is always invertible and the approximate solution is $\hat{x} = (A^TA + \lambda I)^{-1}A^Tb$.

Example 7. In least squares image deblurring we form an estimate $\hat{x}$ by minimizing a cost function of the form $\|Ax - y\|^2 + \lambda(\|D_hx\|^2 + \|D_vx\|^2)$, where D_v and D_h are the vertical and horizontal differencing operations which penalize non-smoothness in the reconstructed image. Suppose the vector x has length MN and contains pixel intensities of an $M \times N$ image X stored column-wise. This penalty term is $\sum_{i=1}^{M}\sum_{j=1}^{N-1}(X_{i,j+1} - X_{ij})^2 + \sum_{i=1}^{M-1}\sum_{j=1}^{N}(X_{i+1,j} - X_{ij})^2$. This implies $\underset{M(N-1) \times MN}{D_h} = \begin{bmatrix} -I & I & 0\ldots0 & 0 & 0 & & \\ 0 & -I & I & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & & -I & I & 0 \\ 0 & 0 & 0 & \ldots & 0 & -I & I \end{bmatrix}$, where all the blocks have size $M \times M$, and $\underset{(M-1)N \times MN}{D_v} = \begin{bmatrix} D & 0 & \ldots & 0 \\ 0 & D & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & D \end{bmatrix}$, where each of the N diagonal blocks D is an $(M-1) \times M$ difference matrix $D = \begin{bmatrix} 1 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & -1 & 1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & -1 & 1 & 0 \\ 0 & 0 & 0 & \ldots & 0 & -1 & 1 \end{bmatrix}$. The regularization term is Dirichlet energy.

```
from PIL import Image
from numpy import asarray
```

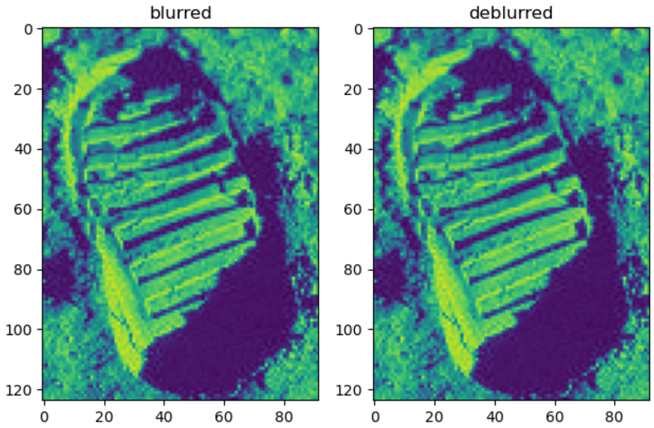

Figure 7: Deblurring of the image.

```
image = Image.open('/home/manish/Pictures/moon.jpg').convert('LA')
im = asarray(image)[:,:,0] / 255
A = np.eye(np.prod(im.shape))
blim = A @ im.flatten() + np.random.randn(A.shape[0])*0.01

M, N = im.shape
Dh = np.zeros((M*(N-1), M*N))
I = np.eye(M)
for i in range(N-1):
    for j in range(N-1):
        if j == i:
            Dh[M*i: min(M*(i+1), Dh.shape[0]), M*j:min(M*(j+1), Dh.shape[1])] = -I
        elif j == i+1:
            Dh[M*i: min(M*(i+1), Dh.shape[0]), M*(j+1):min(M*(j+2), Dh.shape[1])] = I
D = np.zeros((M-1, M))
for i in range(M-1):
    for j in range(M):
        if i == j:
            D[i,j] = -1
        elif j == i+1:
            D[i,j] = 1
Dv = np.zeros(((M-1)*N, M*N))
for i in range(N):
    Dv[(M-1)*i:(M-1)*(i+1),M*i:M*(i+1)] = D

# set up the LS for
# |Ax-b|^2 + l(|Dh x|^2 + |Dv x|^2)
btilde = A.T @ blim
_a, _b, _c = A.T @ A, Dh.T @ Dh, Dv.T @ Dv

from scipy.linalg import lapack

def get_inv(m):
    cholesky, info = lapack.dpotrf(m)
    inv, info = lapack.dpotri(cholesky)
    inv += np.triu(inv, k=1).T
    return inv

l2 = -10
Atilde = _a + 10 ** l2 * _b + 10 ** l2 * _c
a_inv = get_inv(Atilde)
x_hat = a_inv @ btilde
dbim = x_hat.reshape(im.shape)

fig = plt.figure()
ax1 = fig.add_subplot(121)
```

```
plt.imshow(blim.reshape(im.shape))
ax1.set_title('blurred')
ax2 = fig.add_subplot(122)
plt.imshow(dbim)
ax2.set_title('deblurred')
plt.tight_layout()
```

□

Consider fitting data according to the model $\hat{f}(x) = \sum\limits_{i=1}^{p} \theta_i f_i(x)$. Lower values of θ_i are desirable as they decrease sensitivity to a particular basis, except when $f_i(x)$ is constant. This give a bi-criterion objection of $\|y - A\theta\|^2 + \lambda\|\theta_{2:p}\|^2$, where $\lambda > 0$ is the regularization parameter. For a regression model this is written as $\|y - X^T\beta - v\mathbf{1}\|^2 + \lambda\|\beta\|^2$, where the offset v is not penalized. This is also called ridge regression. Highest possible λ for smallest possible RMS is a chosen based on cross-validation. One benefit of adding regularization to the basic least squares data fitting model, that the columns of the associated stacked matrix $\begin{bmatrix} A \\ \sqrt{\lambda}B \end{bmatrix}$ are *always linearly independent*, even if the columns of the matrix A are not. To avoid overfitting features should be kept to 10%-20% of the number of data points. With regularization we can fit a model with more features than would be appropriate without regularization, and this is often the key to success in feature engineering, which can greatly increase the number of features.

In the general case we can solve the least square problem for the stacked $\tilde{A}$ and $\tilde{b}$ in $\mathcal{O}(mn^2)$ time where $m = m_1 + \ldots + m_k$, the sum of the heights of the matrices $A_1, \ldots, A_k$. To solve the problem for L different values of weights, it would take $\mathcal{O}(Lmn^2)$ flops. We start from the formula for the minimizer of the weighted sum objective $\hat{x} = \left(\sum\limits_{i=1}^{k} \lambda_i A_i^T A_i\right)^{-1} \left(\sum\limits_{i=1}^{k} \lambda_i A_i^T b_i\right)$. We can compute $\hat{x}$ by forming these Gram matrices $G_i = A_i^T A_i$ (cost $\mathcal{O}(m_i n^2)$), along with the vectors $h_i = A_i^T b_i$ (cost $\mathcal{O}(2m_i n)$), then forming the weighted sums, and then solving the $n \times n$ set of equations $G\hat{x} = h$. This is a total of $\mathcal{O}(mn^2)$ flops. The weighed sum will be $\mathcal{O}(2kn^2)$ flops and solving $G\hat{x} = h$ is $\mathcal{O}(2n^3)$ flops. *Gram caching* is the simple trick of computing G_i and h_i just once, and reusing these matrices and vectors for the L different choices of weights. This leads to a complexity of $\mathcal{O}(mn^2 + L(k + 2n)n^2)$ flops. Quite often $m >> k + n$, this cost is smaller than the generic $\mathcal{O}(Lmn^2)$ cost.

In many applications $J = \|Ax - b\|^2 + \lambda\|x - x^{des}\|^2$ we have a $m \times n$ matrix A which is wide, i.e. $m < n$ and $\lambda > 0$. The associated $(m + n) \times n$ stacked matrix $\begin{bmatrix} A \\ \sqrt{\lambda}I \end{bmatrix}$ always has linearly independent columns. QR factorization requires $\mathcal{O}(2(m + n)n^2)$ flops. We now use kernel trick to write $\hat{x} = (A^T A + \lambda I)^{-1}(A^T b + \lambda x^{des}) = (A^T A + \lambda I)^{-1} A^T (b - Ax^{des}) + x^{des} = A^T (AA^T + \lambda I)^{-1}(b - Ax^{des}) + x^{des}$. The inverse matrix now has a size $m \times m$. The stacked matrix now is $\begin{bmatrix} A^T \\ \sqrt{\lambda}I \end{bmatrix}$, which has a cost of $\mathcal{O}(2(m + n)m^2)$ flops. This is only linear in n.

5 Constrained least squares

Constrained least square problem can be reduced to a set of linear equations which can be solved via QR factorization efficiently. The problem is

$$\begin{aligned} \text{minimize } & \|\underset{(m\times n)}{A}\underset{(n\times 1)}{x} - \underset{(m\times 1)}{b}\|^2 \\ \text{subject to } & \underset{(p\times n)}{C}\underset{(n\times 1)}{x} = \underset{(p\times 1)}{d}. \end{aligned}$$

This is same as minimizing the weighted objection $\|Ax - b\|^2 + \lambda\|Cx - d\|^2$ for a very large value of λ. A popular example for this is fitting *splines*, which are continuous and differentiable. The other important special case is the least norm problem where we want to minimize $\|x\|^2$ subject to $Cx = d$. The problem can be solved via method of Lagrange multipliers. The KKT equations for the constrained least squares problem can be written as

$$\underset{(n+p)\times(n+p)}{\begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix}} \underset{(n+p)\times 1}{\begin{bmatrix} \hat{x} \\ \hat{z} \end{bmatrix}} = \underset{(n+p)\times 1}{\begin{bmatrix} 2A^T b \\ d \end{bmatrix}},$$

which is a set of $n+p$ linear equations. $\hat{z}$ are the Lagrange multiplier vector. This KKT matrix is invertible iff C *has linearly independent rows, and* $\underset{(n+p)\times n}{\begin{bmatrix} A \\ C \end{bmatrix}}$ *has linearly independent columns.* That is, C is square or wide. The second condition can be satisfied even when the columns of A are linearly independent. We can now solve this via generalization of QR factorization method.

- Compute the QR factorization of $\begin{bmatrix} \underset{m\times n}{A} \\ \underset{p\times n}{C} \end{bmatrix} = \begin{bmatrix} \underset{m\times n}{Q_1} \\ \underset{p\times n}{Q_2} \end{bmatrix} \underset{n\times n}{R}$, and $\underset{n\times p}{Q_2^T} = \underset{(n\times p)}{\tilde{Q}} \underset{(p\times p)}{\tilde{R}}$.
- Compute $\underset{(p\times p)}{\tilde{R}^{-T}} \underset{(p\times 1)}{d}$ by forward substitution.
- Form the right hand side and solve $\underset{(p\times p)}{\tilde{R}} \underset{(p\times 1)}{w} = 2 \underset{(p\times n)}{\tilde{Q}^T} \underset{(n\times m)}{Q_1^T} \underset{(m\times 1)}{b} - 2\tilde{R}^{-T} d$ via back substitution, where $\underset{p\times 1}{w} = \underset{p\times 1}{\hat{z}} - 2 \underset{p\times 1}{d}$.
- Form right hand side and solve $\underset{(n\times n)}{R} \underset{(n\times 1)}{\hat{x}} = \underset{(n\times m)}{Q_1^T} \underset{(m\times 1)}{b} - \frac{1}{2} \underset{(n\times p)}{Q_2^T} \underset{(p\times 1)}{w}$ by back substitution.

This has a complexity of $\mathcal{O}(m+p)n^2$. The least norm problem has the stacked matrix $\begin{bmatrix} I \\ C \end{bmatrix}$. The solution for this problem can be simplified to $\hat{x} = C^T(CC^T)^{-1}d = C^\dagger d$, using the right inverse of C. For $C^T = QR$ as the QR factorization of C^T, we can substitute $C^\dagger = QR^{-1}$ to get $\hat{x} = QR^{-T}d$ as the solution.

Example 8. *(Portfolio Optimization)* A portfolio n-vector w, denoting dollar positions in assets, typically satisfies the allocation rule $\mathbf{1}^T w = 1$. The leverage is defined as $L = \sum_i |w_i|$. Some people use $(L-1)/2$ as the definition of leverage. For multi-period investing held over T periods,we describe the returns by a $T \times n$ matrix R. One of the returns is assumed to be cash with constant returns μ^{rf}. For it being the last column of R as $\mu^{rf}\mathbf{1}$. With re-balancing, the portfolio return in each of the T periods is represented by $r = Rw$. The total value of the portfolio at time t is $V_t = V_1(1+r_1)\ldots(1+r_{t-1}) \approx V_1(1+r_1+\ldots+r_{t-1})$ for small returns. Thus, we have $V_{T+1} \approx V_1 + Tavg(r)V_1$, and we should maximize average returns for maximizing the returns on the portfolio, $E[r]$. This has to balanced against the portfolio risk $std(r)$. Individual assets can be considered portfolios with $w = e_j$. If we fix the return of the portfolio to some given value $\frac{1}{T}\mathbf{1}^T Rw = \mu^T w = \rho$ and minimize variance $\frac{1}{T}\|r - \rho\mathbf{1}\|^2$ we get the problem

$$\begin{aligned} &\text{minimize } \|Rw - \rho\mathbf{1}\|^2 \\ &\text{subject to } \begin{bmatrix} \mathbf{1}^T \\ \mu^T \end{bmatrix} w = \begin{bmatrix} 1 \\ \rho \end{bmatrix}. \end{aligned}$$

The solution to this constrained least square problem is $\begin{bmatrix} w \\ z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 2R^TR & \mathbf{1} & \mu \\ \mathbf{1}^T & 0 & 0 \\ \mu^T & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2\rho^T\mu \\ 1 \\ \rho \end{bmatrix}$. However, we do not know the future returns. If we assume the future asset returns are similar to past returns, we can still use historical average to anticipate future's. Validation here is also called *back-testing.* We use daily return for 19 stocks over 8 years and add a risk free asset with 1% annual return to obtain the 2000 × 20 return matrix R.

```
# minimixe ||Ax - b||^2 s.t. Cx=d
def c_ls(A, b, C, d):
    n = A.shape[0]
    q, r = np.linalg.qr(np.concatenate([A,C], axis=0))
    qt, rt = np.linalg.qr(q[n:].T)
    rmTd = sp.linalg.solve_triangular(rt.T, d, check_finite=False, lower=True)
    w = sp.linalg.solve_triangular(rt, 2 * (qt.T @ q[:n, :].T @ b - rmTd),
                                   check_finite=False, lower=False)
    x_hat = sp.linalg.solve_triangular(r, q[:n, :].T @ b - 0.5 * q[n:, :].T @ w,
                                       check_finite=False, lower=False)
    return x_hat[:,0]
```

```
lev = pd.read_csv(StringIO(data), header=None, sep='\t')
ret = lev.pct_change().round(5).dropna()
eret = np.mean(ret, axis=0)

opt, port_rr = {}, {}
for rho in np.arange(eret.iloc[-1], 0.002, 0.0001):
    # minimize || Rw - rho 1 ||^2  s.t [1^T mu^T]^T w = [1 rho]^T
    w = c_ls(ret, np.ones((ret.shape[0],1)) * rho,
                np.concatenate([np.ones((1,ret.shape[1])), eret.values[None,:]]),
                np.array([[1],[rho]]))
    opt[rho * 250] = w
    port_sigma = np.sqrt(np.mean((ret @ w -rho)**2)*250)
    port_rr[port_sigma] = rho * 250
opt = pd.DataFrame(opt)
ax = pd.Series(port_rr).plot(grid=True)

# asset risk and returns
asset_rr = {}
for i in range(ret.shape[1]):
    ret_a = eret.iloc[i] * 250
    risk_a = ret.iloc[:,i].std() * np.sqrt(250)
    asset_rr[risk_a] = ret_a
pd.Series(asset_rr).plot(ax=ax, style='o', grid=True)
```

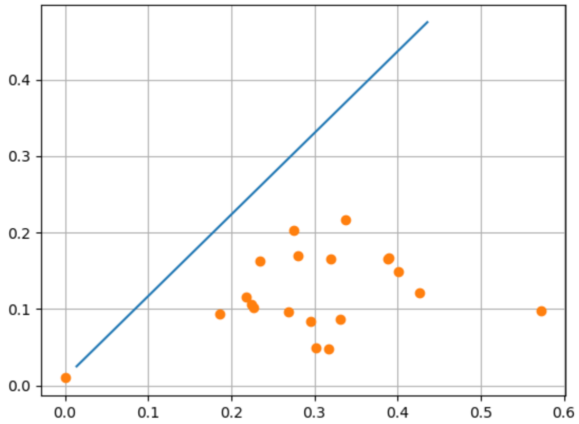

Figure 8: The open circles show the annualized risk and return for 20 assets. The solid line shows the risk and return for the Pareto optimal portfolios.

Figure 8 shows the risk vs return for the 20 assets and the Pareto-optimal risk-return curve which includes a risk free asset. Notice that unconstrained least square will have a slightly different solution and different Pareto optimal curve, when we do not impose $\mathbf{1}^T w = 1$ and $\mu^T w = \rho$.

This formulation can suffer from overfit which can be avoided by regularization, which here means penalizing investments in assets other than cash. This can be done by incorporating $\sum_{i=1}^{n-1} \sigma_i^2 w_i^2$ with a positive multiple λ to the objective. We penalize weights associated with risky assets more than less risky assets.

Further, due to shifts in market we need to update the optimal vector over time over a window M (determined by validation). We can add a regularization term of the form $\kappa \| w^{curr} - w \|^2$ to the objective, where κ is a positive constant and w^{curr} is the currently used allocation, and w is the proposed new allocation vector. This reduces *turnover*, and κ is chosen based on trading costs.

```
def get_matrices_regularized(ret, rho, lam):
    # minimize || Rw - rho 1 ||^2 + lambda ||std w||^2  s.t [1^T mu^T]^T w = [1 rho]^T
    t, n = ret.shape
    eret = np.mean(ret, axis=0)
    std = np.std(ret, axis=0)
    return np.vstack([ret, np.sqrt(lam) * np.diag(std)]), \
           np.vstack([rho * np.ones((t,1)), np.zeros((n, 1))]), \
           np.vstack([np.ones((1,n)), eret.values[None,:]]), \
           np.array([[1],[rho]])

# regularization
rho = 0.1
lam = 10000
opt, port_rr = {}, {}
for rho in np.arange(0, 0.002, 0.0001):
    w = c_ls(*get_matrices_regularized(ret, rho, lam))
    opt[rho * 250] = w
    port_sigma = np.sqrt(np.mean((ret @ w -rho)**2)*250)
    port_rr[port_sigma] = rho * 250
opt = pd.DataFrame(opt)
ax = pd.Series(port_rr).plot(ax=ax,grid=True)
```

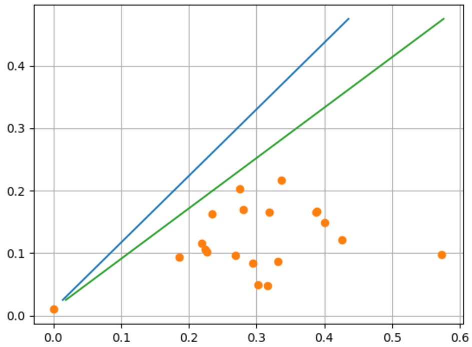

Figure 9: Effect of regularization on the Pareto-optimal curve.

We can express the solution of the portfolio problem in the form

$$\begin{bmatrix} w \\ z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 2R^R & \mathbf{1} & \mu \\ \mathbf{1}^T & 0 & 0 \\ \mu^T & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + \rho \begin{bmatrix} 2R^T R & \mathbf{1} & \mu \\ \mathbf{1}^T & 0 & 0 \\ \mu^T & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2\rho^T\mu \\ 0 \\ 1 \end{bmatrix}.$$

Taking the first n components of this we get $w = w^0 + \rho v$, where w^0 and v are the first n components of the two $(n+2)$ vectors on the right hand side. This shows that the Pareto optimal portfolios form a line in weight space, parametrized by required return ρ. The portfolio w^0 is a point on the line, and the vector v, which satisfies $\mathbf{1}^T v = 0$, gives direction of the line. Thus we can get simply calculate the two vectors and then obtain any optimal portfolio as $w^0 + \rho v$. Thus, all Pareto portfolios are affine combinations of just two portfolios. This is a *two-fund theorem.* □

Example 9. *(Linear quadratic control)* We consider a time-varying linear dynamical system with state n-vector x_t and input m-vector u_t, with dynamics equation $x_{t+1} = A_t x_t + B_t u_t$, $t = 1, 2, \ldots$. The system output is p-vector y_t given by $y_t = C_t x_t$, $t = 1, 2, \ldots$. These variables generally represent some deviations from some standard values and hence they are expected to be small. Linear quadratic control refers to the problem of choosing the input and state sequences, over time period $t = 1, \ldots, T$, so as to minimize sum of the squares objective, subject to the above equations and other constraints, like $x = x^{init}$ and $x_T = x^{des}$. The objective has the form $J = J_{outout} + \rho J_{input}$, where $J_{output} = \sum_{i=1}^{T} \|y_i\|^2 = \sum_{i=1}^{T} \|C_i x_i\|^2$ and $J_{input} = \sum_{j=1}^{T-1} \|u_j\|^2$. The LQC problem is

$$\begin{aligned} \text{minimize} \quad & J_{output} + \rho J_{input} \\ \text{subject to} \quad & x_{t+1} = A_t x_t + B_t u_t, \ t = 1, \ldots, T-1, \\ & x_1 = x^{init}, \qquad x_T = x^{des}, \end{aligned}$$

where the variable to be chosen are $z = (x_1, \dots, x_T, u_1, \dots, u_{T-1})$ of size $Tn + (T-1)m$. We can write it as $\|\tilde{A}z - \tilde{b}\|^2$ with $b = 0$

$$\tilde{A} = \begin{bmatrix} C_1 & & & & & & \\ & C_2 & & & & & \\ & & \ddots & & & & \\ & & & C_T & & & \\ & & & & \sqrt{\rho} I_m & & \\ & & & & & \ddots & \\ & & & & & & \sqrt{\rho} I_m \end{bmatrix}$$

The dynamics constraints, the the initial and final state constraints, can be expressed as $\tilde{C}z = \tilde{d}$, with

$$\tilde{C} = \begin{bmatrix} A_1 & -I_n & & & & B_1 & & & \\ & A_2 & -I_n & & & & B_2 & & \\ & & \ddots & \ddots & & & & \ddots & \\ & & & A_{T-1} & -I_n & & & & B_{T-1} \\ I_n & & & & & & & & \\ & & & & I_n & & & & \end{bmatrix}, \quad \tilde{d} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ x^{init} \\ x^{des} \end{bmatrix}$$

The solution $\hat{z}$ of the constrained least square problem gives us the optimal trajectory and the associated optimal state trajectory. Consider the time-invariant linear dynamical system with $A = \begin{bmatrix} 0.855 & 1.61 & 0.667 \\ 0.015 & 1.073 & 0.053 \\ -0084 & 0.059 & 1.022 \end{bmatrix}$, $B = \begin{bmatrix} -0.076 \\ -0.0139 \\ 0.342 \end{bmatrix}$, and $C = \begin{bmatrix} 0.218 & -3.597 & -1.683 \end{bmatrix}$, with initial conditions $x^{init} = (0.496, -0.745, 1.394)$, target final state of $x^{des} = 0$ and $T = 100$. Both y_t and u_t are scalars in this case. When the input is zero, the open-loop outout is given by $y_t = CA^{t-1}x^{init}$, $t = 1, \dots, T$ as can be easily ascertained by repeated substitution. We use $\rho = 0.05, 0.2, 1$ to show the trajectories in this case. Increasing ρ has effect of decreasing J_{input}, at the cost of increasing J_{output}.

```
# data
A = np.array([[0.855, 1.161, 0.667],[0.015, 1.073, 0.053],[-0.084, 0.059, 1.022]])
B = np.array([[-0.076, -0.139, 0.342]]).T
C = np.array([[0.218, -3.597, -1.683]])
x_init = np.array([[0.496, -0.745, 1.394]]).T
x_des = 0 * x_init
T = 100

def get_matrices(rho):
    n , m = B.shape
    p , n = C.shape
    l = T*n + (T-1)*m
    A_tilde = np.zeros((T*p+(T-1)*m, l))
    for i in range(T):
        A_tilde[i*p:(i+1)*p, i*n:(i+1)*n] = C
        if i < T-1:
            A_tilde[T*p+i*m:T*p+(i+1)*m, T*n+i*m:T*n+(i+1)*m] = np.eye(m) * np.sqrt(rho)
    b_tilde = np.zeros((A_tilde.shape[0], 1))
    C_tilde = np.zeros((n*(T+1), n*T+(T-1)*m))
    for i in range(T-1):
        C_tilde[i*n:(i+1)*n, i*n:(i+1)*n] = A
        C_tilde[i*n:(i+1)*n, (i+1)*n:(i+2)*n] = - np.eye(n)
        C_tilde[i*n:(i+1)*n, T*n+i*m:T*n+(i+1)*m] = B
    C_tilde[(T-1)*n:T*n, :n] = np.eye(n)
    C_tilde[T*n:(T+1)*n, (T-1)*n:T*n] = np.eye(n)
    d_tilde = np.vstack([np.zeros((n*(T-1),1)), x_init, x_des])
    # print(A_tilde.shape, b_tilde.shape, C_tilde.shape, d_tilde.shape)
    return A_tilde, b_tilde, C_tilde, d_tilde
```

```
# analysis
n, _ = A.shape
y_open_loop = np.zeros((T,))
a = np.eye(n)
for t in range(T):
    a = a @ A
    y_open_loop[t] = (C @ a @ x_init)[0][0]

df = pd.DataFrame({
    'y_ol': y_open_loop,
})
df.plot(grid=True)

# pareto optimal curve
po, u, y = {}, {}, {}
for rho in np.arange(0.01, 1.001, 0.01):
    x_hat = c_ls(*get_matrices(rho))
    y[rho] = (C @ x_hat[:T*n].reshape((T, n)).T)[0]
    u[rho] = x_hat[T*n:]
    j2 = np.sum(y[rho]**2)
    j1 = np.sum(u[rho]**2)
    po[j1] = j2
pd.Series(po).plot(grid=True, title='Pareto optimal curve')
pd.DataFrame({'0.05': u[0.05], '0.2': u[0.2], '1': u[1]}).plot(grid=True, title='u')
pd.DataFrame({'0.05': y[0.05], '0.2': y[0.2], '1': y[1]}).plot(grid=True, title='y')
```

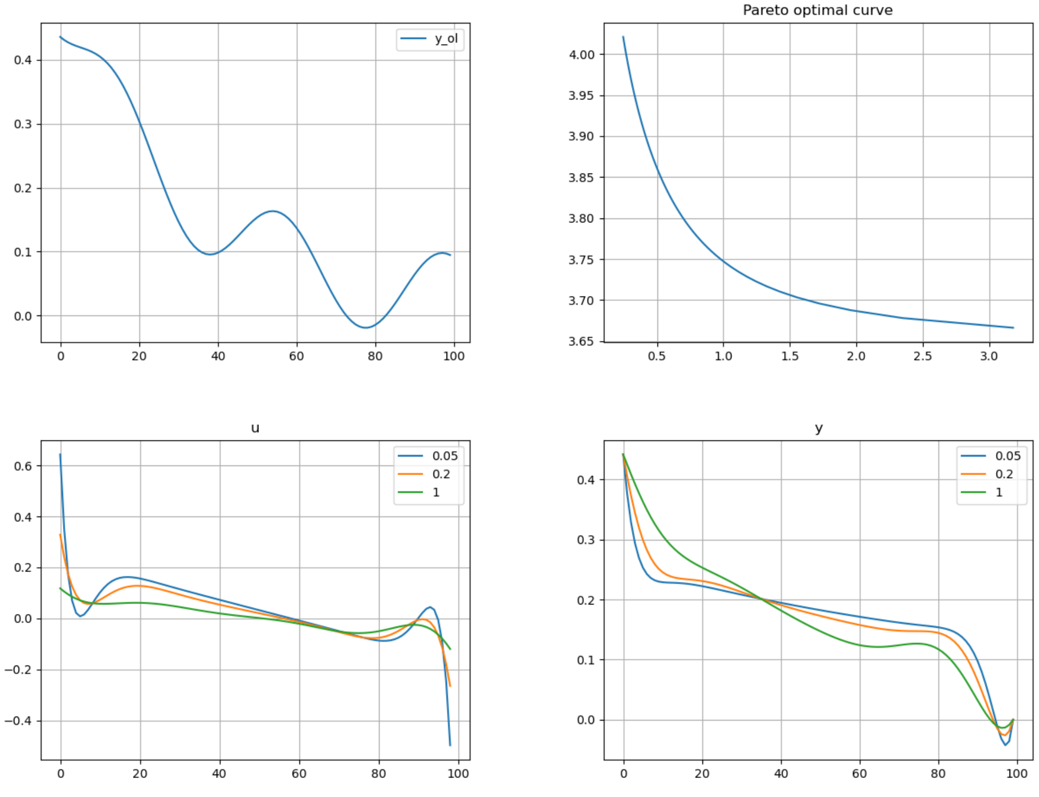

Figure 10: Open loop evolution, Pareto optimal curve, optimal inputs and outputs.

We now look at a few variations. For **tracking** problem we replace y_t in J_{output} (called tracking error) with $y_t - y_t^{des}$, where y_t^{des} is given desired output trajectory. Decreasing ρ leads to better output tracking, at the cost of larger input trajectory. The desired trajectory y_t^{des} appears in the vector $\tilde{b}$. We can also give **time-weighted** output by using $J_{output} = \sum_{i=1}^{T} w_i \|y_i\|^2$, where $w_1, \ldots, w_T$ are given positive constants, e.g. exponential weighting $w_t = \theta^t$, where $\theta > 0$. If $\theta < 1$ it is also called discount factor. Finally, we can also specify **way-point constraints** like $y_\tau = y^{wp}$, where y^{wp} is a given p-vector, and τ is a given way-point time. This is expressed as linear equality constraints on the big vector z.

In *linear state feedback control* we measure the state in each period and use it as an input $u_t = \underset{m\times n}{K} x_t, t = 1, \ldots, T$ where K is called the *state feedback gain matrix.* This is generally used for infinite horizon problem where we want both x_t and u_t to converge to zero. To find K we note that for state feedback the solution $\hat{z}$ is a linear function of x^{init} as we assume $x^{des} = 0$. Since $\hat{u}_1$, the optimal input at $t = 1$, is a linear function of x_1, $u_1 = Kx_1$. The columns of K can be found by solving the LQC problem with initial conditions $x^{init} = e_1, \ldots, e_n$. This can be done efficiently by factoring the coefficient matrix once, and then carrying out n solves. With this choice of K, the input u_1 with state feedback control and under linear quadratic control are the same; for $t > 1$ the two inputs differ. This K is not very much dependent on T, provided it is chosen large enough.

```
# state feedback control
rho = 1

a, b, c, d = get_matrices(rho)

n , m = B.shape
N = a.shape[0]
q, r = np.linalg.qr(np.vstack([a,c]))
qt, rt = np.linalg.qr(q[N:].T)

K = np.zeros((m, n))
for i in range(n):
    xinit = np.zeros((n,1))
    xinit[i,0] = 1
    d[-6:-3] = xinit
    rmTd = scla.solve_triangular(rt.T, d, check_finite=False, lower=True)
    w = scla.solve_triangular(rt, 2 * (qt.T @ q[:N, :].T @ b - rmTd),
                              check_finite=False, lower=False)
    x_hat = scla.solve_triangular(r, q[:N, :].T @ b - 0.5 * q[N:, :].T @ w,
                              check_finite=False, lower=False)
    K[:,i] = x_hat[T*n]

K
>> array([[ 0.30832877, -2.65864963, -1.44602291]])

# find LQC and SFC trajectories for u and y
n, _ = A.shape
u_sf = np.zeros((2*T-1,))
y_sf = np.zeros((2*T,))
a = np.eye(n)
for t in range(2*T):
    a = a @ (A + B @ K)
    if t<2*T-1:
        u_sf[t] = (K @ a @ x_init)[0][0]
    y_sf[t] = (C @ a @ x_init)[0][0]

pd.DataFrame({'State feedback': u_sf, 'Optimal': np.hstack([u[1],
                np.zeros(100)])}).plot(grid=True, title='u')
pd.DataFrame({'State feedback': y_sf, 'Optimal': np.hstack([y[1],
                np.zeros(100)])}).plot(grid=True, title='y')
```

□

Example 10. *(Linear quadratic state estimation)* The setting is a linear dynamical system of the form $x_{t+1} = A_t x_t + B_t w_t$, $y_t = C_t x_t + v_t$, $t = 1, 2, \ldots$, where $\underset{n\times 1}{x_t}$ is the state of the system, $\underset{p\times 1}{y_t}$ is the measurement, $\underset{m\times 1}{w_t}$ is the input or process noise, and $\underset{p\times 1}{v_t}$ is the measurement residual. In state estimation, we know the matrices A_t, B_t, C_t over the time period $t = 1, \ldots, T$ as well as the measurement $y_1, \ldots, y_T$, but we do not know the process or measurement noises. The goal is to estimate the state space sequence $x_1, \ldots, x_T$. The measurement noise $v_t = y_t - C_t x_t$ is assumed to be small. Thus our objective is to minimize the residual $J_{meas} = \|v_1\|^2 + \ldots + \|v_T\|^2 = \|C_1 x_1 - y_1\|^2 + \ldots + \|C_T x_T - y_T\|^2$. The secondary objective is the sum of squares

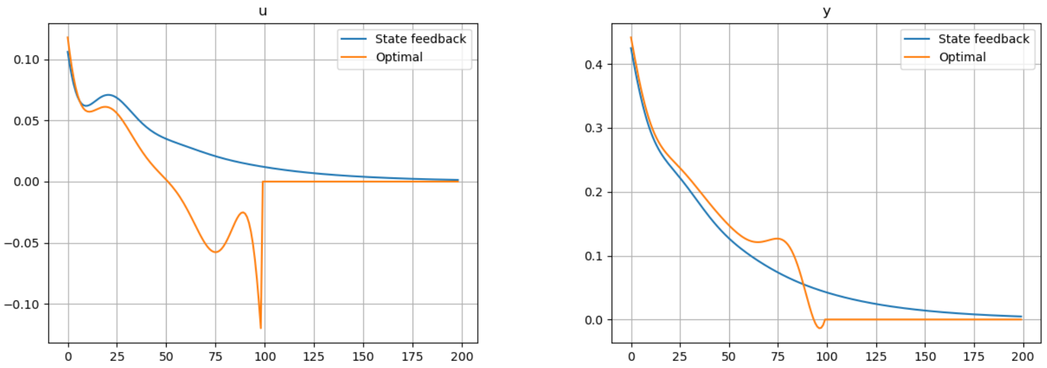

Figure 11: The curves show the solution for $\rho = 1$ for linear quadratic control and simpler linear state control $u_t = K x_t$.

of the norms of the process noise $J_{proc} = \|w_1\|^2 + \ldots + \|w_{T-1}\|^2$. We thus solved the problem

$$\begin{array}{c} \text{minimize } J_{means} + \lambda J_{proc} \\ \text{subject to } x_{t+1} = A_t x_t + B_t w_t, \quad t = 1, \ldots, T-1, \end{array}$$

When λ is small it means we trust the measurement more. This problem is similar to the linear quadratic control problem with some differences. There would could chose the inputs, here the inputs are unknown and the problem is to estimate them. This connection is called *control/estimation duality*. We define the state vector $z = (x_1, \ldots, x_T, w_1, \ldots, w_{T-1})$. The objective can be expressed as $\|\tilde{A}z - \tilde{b}\|^2$, with

$$\tilde{A} = \begin{bmatrix} C_1 & & & & & & \\ & C_2 & & & & & \\ & & \ddots & & & & \\ & & & C_T & & & \\ & & & & \sqrt{\lambda} I_m & & \\ & & & & & \ddots & \\ & & & & & & \sqrt{\lambda} I_m \end{bmatrix}, \quad \tilde{b} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_T \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The constraints can be expressed as $\tilde{C}z = \tilde{d}$, with $\tilde{d} = 0$ and

$$\tilde{C} = \begin{bmatrix} A_1 & -I_n & & & & B_1 & & & \\ & A_2 & -I_n & & & & B_2 & & \\ & & \ddots & \ddots & & & & \ddots & \\ & & & A_{T-1} & -I_n & & & & B_{T-1} \end{bmatrix}.$$

The recursive method to solve the same problem is called *Kalman filtering*. We consider a motion of mass moving in 2d. The first two components of x_t represent the position coordinates, and the last two component represent the velocity components. The input w_t acts like a force on the mass. Cx_t is a 2-vector representing the true position of the mass at time t. The measurement $y_t = Cx_t + v_t$ is a noisy measurement of the mass position. We want to estimate the trajectory over $t = 1, \ldots, T$ with $T = 100$.

```
from ECONOMETRICS.Boyd.lqse_data import lq_estimation_data
y = lq_estimation_data()

A = np.array([[1,0,1,0],[0,1,0,1],[0,0,1,0],[0,0,0,1]])
B = np.array([[0,0],[0,0],[1,0],[0,1]])
C = np.array([[1,0,0,0],[0,1,0,0]])

# solve
# SUM ||C_i x_i - y_i ||^2 + lambda X SUM || w_j ||^2
# s.t. x_{t+1} = A x_t + B_t w_t, for t = 1, ..., T-1
```

```
def get_matrices(lam):
    n, m = B.shape
    p, n = C.shape
    A_tilde = np.zeros((T*p+(T-1)*m, T*n+(T-1)*m))
    for i in range(T):
        A_tilde[p*i:p*(i+1),n*i:n*(i+1)] = C
        if i < T-1:
            A_tilde[T*p + m*i: T*p + m*(i+1), T*n+m*i:T*n+m*(i+1)] = np.sqrt(lam) * np.eye(m)
    b_tilde = np.zeros((T*p+(T-1)*m,1))
    for i in range(T):
        b_tilde[p*i:p*(i+1)] = y[:,i:i+1]
    C_tilde = np.zeros(((T-1)*n, T*n+(T-1)*m))
    for i in range(T-1):
        C_tilde[n*i:n*(i+1), n*i:n*(i+1)] = A
        C_tilde[n*i:n*(i+1), n*(i+1):n*(i+2)] = -np.eye(n)
        C_tilde[n*i:n*(i+1), T*n+m*i:T*n+m*(i+1)] = B
    d_tilde = np.zeros(((T-1)*n, 1))
    print(A_tilde.shape, b_tilde.shape, C_tilde.shape, d_tilde.shape)
    return A_tilde, b_tilde, C_tilde, d_tilde

x_hat = {}
for lam in [1, 1e3, 1e5]:
    x_hat[lam] = c_ls(*get_matrices(lam))

# analysis
ax=pd.DataFrame(y.T).plot.scatter(x=0, y=1, grid=True, color='g')
pd.DataFrame(x_hat[1][:T*n].reshape((T, n)) @ C.T).
            plot(ax=ax,x=0, y=1, grid=True, style='-', color='r')
pd.DataFrame(x_hat[1e3][:T*n].reshape((T, n)) @ C.T).
            plot(ax=ax,x=0, y=1, grid=True, style='-', color='b')
pd.DataFrame(x_hat[1e5][:T*n].reshape((T, n)) @ C.T).
            plot(ax=ax,x=0, y=1, grid=True, style='-', color='k')
ax.legend(['1', '1e3', '1e5', 'measurement'])
plt.tight_layout()
```

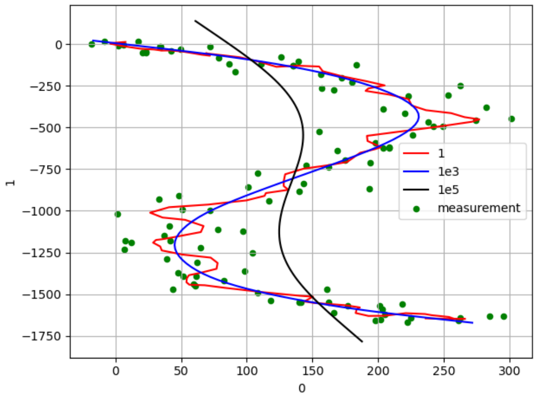

Figure 12: The circles show the 100 noisy measurement in 2D. The solid likes show the estimated trajectories $C\hat{x}_t$ for the three values of λ.

There are few variations we can consider for the state estimation problem. If we add the constraint $x_1 = x_1^{known}$ we account for the known initial state x_1. If we know y_t for $t \in \mathcal{T}$, where $\mathcal{T}$ is the set of times for which we have a measurement (i.e., there are missing measurements), we can replace $\sum_{t=1}^{T} \|v_t\|^2$ with $\sum_{t \in \mathcal{T}} \|v_t\|^2$. We can estimate the missing measurements by taking $\hat{y}_t = C_t\hat{x}_t$, $t \notin \mathcal{T}$. This directly gives us a method to validate a quadratic state estimation method to choose λ. To do this, we remove some of the measurements, say 20%, and carry out least squares state estimation pretending that those measurements are missing. Our state estimate procedures predicted values for the missing measurements, which we can compare to the actual measurements. We choose a value of λ which approximately minimizes this test prediction error.

We change the value of $J_{means} = \sum_{t\in\mathcal{T}} \|Cx_t - y_t\|^2$ and calculate train and test errors to choose the best λ which comes out to around 10^3.

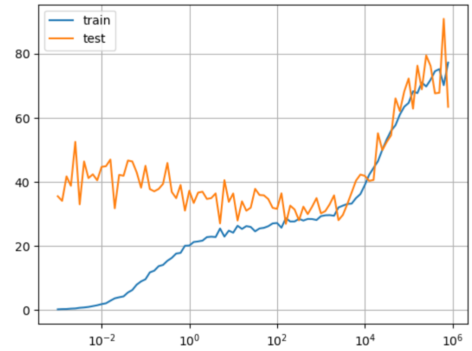

Figure 13: Training and test errors for the state estimation example with different random choices of 20% test set for each λ.

```
# validation
def get_matrices_val(lam, test_frac):
    n, m = B.shape
    p, n = C.shape
    TT = int(T*(1-test_frac))
    # np.random.seed(0)
    train = sorted(np.random.choice(range(100), TT, replace=False))
    A_tilde = np.zeros((T*p+(T-1)*m, T*n+(T-1)*m))
    for i in range(T):
        if i in train:
            A_tilde[p*i:p*(i+1),n*i:n*(i+1)] = C
        if i < T-1:
            A_tilde[T*p + m*i: T*p + m*(i+1), T*n+m*i:T*n+m*(i+1)] = np.sqrt(lam) * np.eye(m)
    b_tilde = np.zeros((T*p+(T-1)*m,1))
    for i in range(T):
        if i in train:
            b_tilde[p*i:p*(i+1)] = y[:,i:i+1]
    C_tilde = np.zeros(((T-1)*n, T*n+(T-1)*m))
    for i in range(T-1):
        C_tilde[n*i:n*(i+1), n*i:n*(i+1)] = A
        C_tilde[n*i:n*(i+1), n*(i+1):n*(i+2)] = -np.eye(n)
        C_tilde[n*i:n*(i+1), T*n+m*i:T*n+m*(i+1)] = B
    d_tilde = np.zeros(((T-1)*n, 1))
    return A_tilde, b_tilde, C_tilde, d_tilde, train

def calc_rms(resid):
    n, p = resid.shape
    return np.sqrt(np.sum(resid**2)/n/p)

rms = {}
for lp in np.arange(-3, 6, 0.1):
    a, b, c, d, train = get_matrices_val(10**lp, 0.2)
    test = [i for i in range(T) if i not in train]
    x_hat = c_ls(a, b, c, d)
    y_hat = x_hat[:T * n].reshape((T, n)) @ C.T
    rms[10**lp] = pd.Series({'train': calc_rms(y_hat[train, :] - y.T[train,:]),
                              'test': calc_rms(y_hat[test, :] - y.T[test, :])})
pd.DataFrame(rms).T.plot(grid=True, logx=True)
```

□

6 Nonlinear least squares

If we have nonlinear functions we can still go a long with some simple heuristic. Consider a set of m possibly nonlinear equations in n unknowns $x = (x_1, \dots, x_n)$, written as $f_i(x) = 0$, $i = 1, \dots, m$, where $f_i : \boldsymbol{R}^n \to \boldsymbol{R}$ is a scalar valued function. We refer to $f_i(x) = 0$ as the ith residual. We write it in compact form as $f(x) = 0$, where $f(x) = (f_1(x), \dots, f_m(x))$ is an m-vector, $f : \boldsymbol{R}^n \to \boldsymbol{R}^m$. If f is affine, we can use the linear methods we covered in the previous sections. When we cannot find an exact solution we seek an approximate solution. We seek to solve the *nonlinear least squares problem*

$$\text{minimize } \|f(x)\|^2,$$

where n-vector x is the variable to be found. The optimality condition is $\frac{\partial}{\partial x_i}\|f(\hat{x})\|^2 = 0$, $i = 1, \dots, n$. In vector form $\nabla\|f(\hat{x})\|^2 = 0$ or $\underset{m\times n}{Df(\hat{x})}^T \underset{m\times 1}{f(\hat{x})} = \underset{n\times 1}{0}$, where $Df(x)$ is the Jacobian matrix of the function f at the point x. This is just a *necessary* condition, as this condition can be true for other non solution points too. This is a hard problem to solve, so we will settle for heuristics, that are not guaranteed to give a solution, but often do.

6.1 Gauss-Newton algorithm

We describe Gauss-Newton algorithm here which is iterative in nature and generate a sequence of points $x^{(1)}, x^{(2)}, \dots$. The vector $x^{(1)}$ is called the starting point of the algorithm, and $x^{(k)}$ is called the kth iterate. The algorithm is terminated when the residual $\|f(x^{(k)})\|$ is small enough, or $x^{(k+1)}$ is very close to $x^{(k)}$, or when a maximum number of iterations is reached.

At each iteration k, we form the affine approximation $\hat{f}$ of f at the current iterate $x^{(k)}$, given by Taylor approximation

$$\hat{f}(x; x^{(k)}) = f(x^{(k)}) + \underbrace{Df(x^{(k)})}_{m\times n}(x - x^{(k)}).$$

The affine function $\hat{f}(x; x^{(k)})$ is a very good approximation of $f(x)$ provided x is near $x^{(k)}$, i.e. $\|x - x^{(k)}\|$ is small.

The next iterate $x^{(k+1)}$ is then taken to be the minimizer of $\|\hat{f}(x; x^{(k)})\|^2$. Assuming the Jacobian matrix $Df(x^{(k)})$ has linearly independent columns $(m \geq n)$ we have

$$x^{(k+1)} = x^{(k)} - \left(Df(x^{(k)})^T Df(x^{(k)})\right)^{-1} Df(x^{(k)})^T f(x^{(k)}).$$

The algorithm is terminated early if $f(x)$ is very small, or $x^{(k+1)} \approx x^{(k)}$; and it terminates with error if the columns of $Df(x^{(k)})$ are linearly dependent. The natural stopping point always satisfies $Df(x^k)Tf(x^k) = 0$, i.e. when the optimality condition holds. The norm of the residual of the approximation goes down in each iteration, $\|\hat{f}(x^{(k+1)}; x^{(k)})\|^2 \leq \|f(x^{(k)})\|^2$ which is not same as $\|f(x^{(k+1)})\|^2 \leq \|f(x^{(k)})\|^2$, the norm of the residual going down which is what we would like.

For the special case of $m = n$, the algorithm is called Newton-Raphson algorithm. When $m = n$, the Jacobian is a square, so the basic Gauss-Newton update can be simplified to $x^{(k+1)} = x^{(k)} - \left(Df(x^{(k)})\right)^{-1} f(x^{(k)})$. For $n = 1$ it is called Newton algorithm and the update is written as $x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$.

Gauss-Newton can work well, i.e. $x^{(k)}$ very quickly converges to a point with small residual. But it has two serious shortcomings:

- Diverge: If the approximation $\|f(x)\| \approx \|\hat{f}(x; x^{(k)})\|^{22}$ is not true when $x^{(k+1)}$ is not near $x^{(k)}$, the true residual at $x^{(k+1)}$ can be larger than the residual at $x^{(k)}$ and the algorithm can diverge.
- Fails: When the assumption that the columns of the Jacobian matrix $Df(x^{(k)})$ are linearly independent does not hold at any iteration, $x^{(k+1)}$ is not defind and the algorithm fails and stops.

6.2 Levenberg-Marquardt algorithm

We now address the shortcomings we just described using the Levenberg-Marquardt algorithm. We choose two objectives: $\|\hat{f}(x; x^{(k)})\|^2$ small, and $\|x - x^{(k)}\|^2$ small. The second objective keeps the next iterate close enough so that we can trust the affine approximation. Thus, we should choose $x^{(k+1)}$ as the minimizer of $\|\hat{f}(x; x^{(k)})\|^2 + \lambda^{(k)}\|x - x^{(k)}\|^2$, where $\lambda^{(k)}$ is a positive parameter and can take different values in different iterations. For $\lambda^{(k)} = 0$ we get back the Gauss-Newton algorithm. The second term is called *trust penalty* and $\lambda^{(k)}$ is called *trust parameter*. We, thus, get the update rule as

$$x^{(k+1)} = x^{(k)} - \left(Df(x^{(k)})^T Df(x^{(k)}) + \lambda^{(k)} I\right)^{-1} Df(x^{(k)})^T f(x^{(k)}).$$

The matrix inverse here always exists. Thus both the shortcomings are addressed. The algorithms stops only when the optimality conditions are satisfied. When $\lambda^{(k)}$ is too small $x^{(k+1)}$ can be far enough away from $x^{(k)}$, while when $\lambda^{(k)}$ is too large it may take too many iterations to make progress. We use the following heuristics to choose $\lambda^{(k)}$:

- Compute tentative iterate $x^{(k+1)}$ minimizer of $\|\hat{f}(x; x^{(k)})\|^2 + \lambda^{(k)}\|x - x^{(k)}\|^2$.
- If $\|f(x^{(k+1)})\|^2 < \|f(x^{(k)})\|^2$, accept the iterated and reduce λ: $\lambda^{(k+1)} = 0.8\lambda^{(k)}$. Otherwise, increase λ and do not update x: $\lambda^{(k+1)} = 2\lambda^{(k)}$ and $x^{(k+1)} = x^{(k)}$.

We look at some practical aspects of this algorithm.

- Stopping criteria - we stop before k^{max} if we reach a small enough residual $\|f(x^{(k+1)})\|^2$ or a small enough optimality condition residual $\|Df(\hat{x})^T f(\hat{x})\|^2$. For the latter condition it could or could not be a minimizer of $\|f(x)\|^2$.
- Warm start - When a sequence of similar or related nonlinear least square problems are solved, we can start the LM algorithm at the solution of the previously solved problem, which can greatly reduce the number of iterations required to converge.
- Multiple runs - It is common to run LM algorithm for various starting points $x^{(1)}$. We then use the best one found with the smallest $\|f(x)\|^2$.
- Complexity - The first step of Jacobian calculation depends on the form of f, the second step is $\mathcal{O}(2(m + n)n^2)$ flops.
- LM update for $n = 1$ is given by $x^{(k+1)} = x^{(k)} - \frac{f'(x^{(k)})}{\lambda^{(k)} + (f'(x^{(k)}))^2} f(x^{(k)})$.

Example 11. *(Nonlinear equation)* We look at the Sigmoid function $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ which has a unique zero at the origin. We start at $x^{(1)} = 0.95$ and $x^{(1)} = 1.15$ to see how the various algorithms behaves.

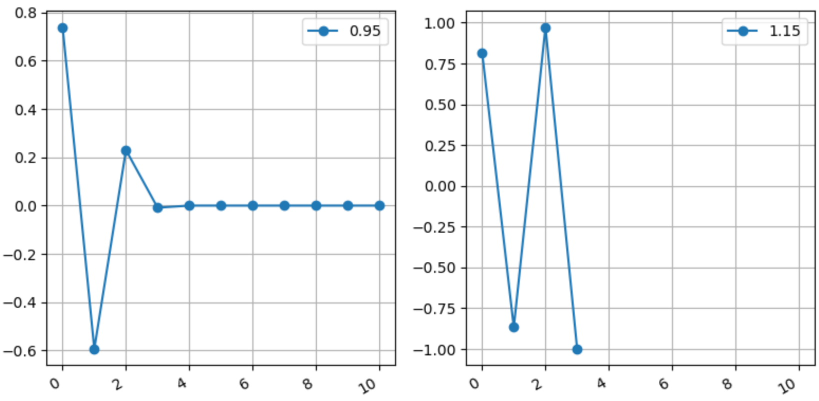

Figure 14: Value of $f(x^{(k)})$ versus iteration number k for Gauss-Newton method for $x^{(1)} = 0.95$ and $x^{(1)} = 1.15$.

```
def f(x):
    e2x = np.exp(2*x)
    return (e2x-1)/(e2x+1)

def jac(x):
    e2x = np.exp(2*x)
    return 4 * e2x / (e2x + 1)**2

# GN
def gn(x, count_max=20, err=1e-8):
    count = 0
    fx_arr = [f(x)]
    while (count < count_max) or (np.abs(f(x)) > err):
        x = x - f(x)/jac(x)
        fx_arr.append(f(x))
        count += 1
    return fx_arr

pd.DataFrame({'0.95': gn(0.95, 10), '1.15': gn(1.15, 10)}).\
    plot(grid=True, style='o-', subplots=True, layout=(1,2), figsize=(8,4))
plt.tight_layout()
```

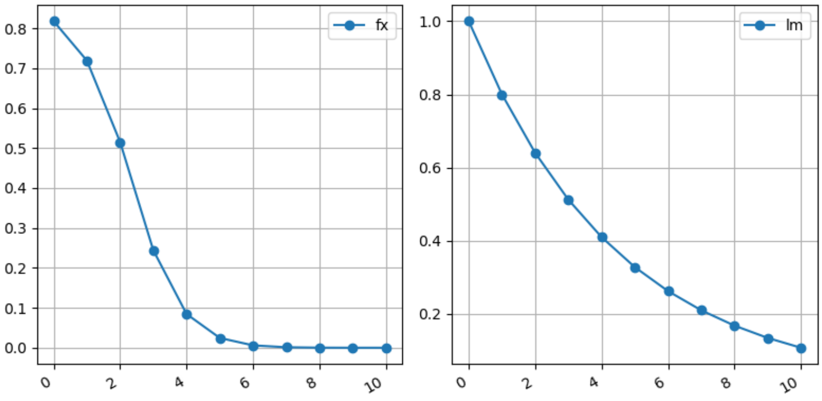

Figure 15: Values of $f(x^{(k)})$ and $\lambda^{(k)}$ versus the iteration number k for the Levenberg-Marquardt algorithm applied to sigmoid function with starting point $x^{(1)} = 1.15$ and $\lambda^{(1)} = 1$.

```
# LM
def lm(x, lam=1, count_max=20, err=1e-8):
    count = 0
    fx_arr = [f(x)]
    lam_arr = [lam]
    while (count < count_max) and (np.abs(f(x)) > err):
        j = jac(x)
        _x = x - f(x) * j /(lam + j**2)
        if np.abs(f(_x)) < np.abs(fx_arr[-1]):
            lam = 0.8 * lam
            x = _x
        else:
            lam = 2 * lam
        fx_arr.append(f(x))
        lam_arr.append(lam)
        count += 1
    return fx_arr, lam_arr

fx, lam = lm(1.15, 1, 10)
pd.DataFrame({'fx': fx, 'lm':lam}).plot(subplots=True,
            figsize=(8,4), layout=(1,2), grid=True, style='o-')
plt.tight_layout()
```

□

Example 12. *(Equilibrium prices)* Here we investigate the equilibrium price problem, with supply and demand functions

$$D(p) = \exp\left(E^d(\log p - \log p^{nom}) + d^{nom}\right)$$
$$S(p) = \exp\left(E^s(\log p - \log p^{nom}) + s^{nom}\right)$$

where E^d and E^s are demand and supply elasticity matrices, d^{nom} and s^{nom} are the nominal demand and supply vectors, and the log and exp are applied element wise. The residual is $f(p) = S(p) - D(p)$, the excess supply. When this residual is minimized we uncover the equilibrium price p^*. We show the code to solve the problem for the given data. The $f(p)$ 2-vector is given by $\begin{bmatrix} e^{E^s_{00}\log p_1 + E^s_{01}\log p_2 + s_0^{nom}} - e^{E^d_{00}\log p_1 + E^d_{01}\log p_2 + d_0^{nom}} \\ e^{E^s_{10}\log p_1 + E^s_{11}\log p_2 + s_1^{nom}} - e^{E^d_{11}\log p_1 + E^d_{11}\log p_2 + d_1^{nom}} \end{bmatrix} = \begin{bmatrix} e^{S^0} - e^{D^0} \\ e^{S^1} - e^{D^1} \end{bmatrix}$ and thus the Jacobian matrix $Df(x)$ is 2×2 matrix $\begin{bmatrix} \frac{1}{p_1}(e^{S^0}E^s_{00} - e^{D^0}E^d_{00}) & \frac{1}{p_2}(e^{S^0}E^s_{01} - e^{D^0}E^d_{01}) \\ \frac{1}{p_1}(e^{S^1}E^s_{10} - e^{D^1}E^d_{10}) & \frac{1}{p_2}(e^{S^1}E^s_{11} - e^{D^1}E^d_{11}) \end{bmatrix}$

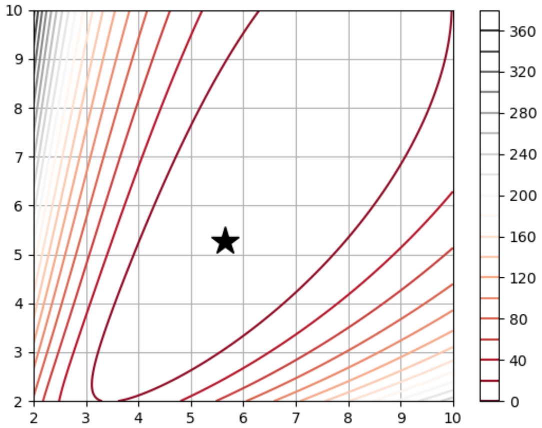

Figure 16: Contour lines of the square norm of the excess supply for a small example of two commodities. The point marked star is the equilibrium prices for with $f(p) = 0$.

```
p_nom = np.array([[2.8, 10]]).T
d_nom = np.array([[3.1, 2.2]]).T
s_nom = np.array([[2.2, 0.3]]).T
Ed = np.array([[-0.5, 0.2],[0, -0.5]])
Es = np.array([[0.5, -0.3],[-0.15, 0.8]])

def f(p):
    dp = np.log(p)
    return np.exp(Es @ dp + s_nom) - np.exp(Ed @ dp + d_nom)

def jacobian(p):
    p1, p2 = p.ravel()
    dp = np.log(p)
    S0, S1 = np.exp(Es @ dp + s_nom).ravel()
    D0, D1 = np.exp(Ed @ dp + d_nom).ravel()
    return np.array([[(S0*Es[0,0]-D0*Ed[0,0])/p1, (S0*Es[0,1]-D0*Ed[0,1])/p2],
                    [(S1*Es[1,0]-D1*Ed[1,0])/p1, (S1*Es[1,1]-D1*Ed[1,1])/p2]])

p0 = np.array([3, 9])
l0 = 1

# contour
p1 = np.linspace(2, 10, 1000)
p2 = np.linspace(2, 10, 1000)
P1, P2 = np.meshgrid(p1, p2)
p = np.vstack([P1.ravel(), P2.ravel()])
Z = np.sum(f(p)**2, axis=0).reshape((1000,1000))
```

```
ax = plt.contour(P1, P2, Z, 20, cmap='RdGy')
ax.axes.grid(True)
x, y = np.where(Z==np.min(Z))
x, y = x[0], y[0]
ax.axes.plot(P1[x,y], P2[x,y], '*', color='k', markersize='20')
plt.colorbar()
```

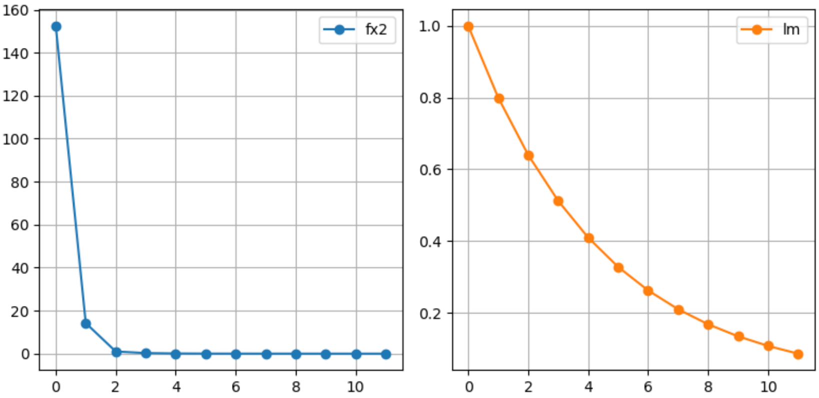

Figure 17: Clost function $\|f(p^{(k)})\|^2$ and the trust parameter $\lambda^{(k)}$ versus iteration number k for the price equilibrium example.

```
# RLS
def rls(A, b, lam):
    gm = A.T @ A + lam * np.eye(A.shape[1])
    q, r = np.linalg.qr(gm)
    A_inv = scla.solve_triangular(r, q.T, check_finite=False, lower=False)
    return A_inv @ b

# LM
def lm(x, lam=1, count_max=20, err=1e-4):
    count = 0
    fx_arr = [np.sum(f(x)**2)]
    lam_arr = [lam]
    while (count < count_max) and (np.sum(f(x)**2) > err):
        j = jacobian(x)
        _x = x - rls(j, j.T @ f(x), lam)
        if np.sum(f(_x)**2) < fx_arr[-1]:
            lam = 0.8 * lam
            x = _x
        else:
            lam = 2 * lam
        fx_arr.append(np.sum(f(x)**2))
        lam_arr.append(lam)
        count += 1
    return fx_arr, lam_arr, x


fx, lam, x_star = lm(np.array([[3,9]]).T, 1)
pd.DataFrame({'fx2': fx, 'lm':lam}).plot(subplots=True,
            figsize=(8,4), layout=(1,2), grid=True, style='o-')
plt.tight_layout()
```

□

Example 13. *(Location from range measurement)* We look at the problem of estimating location from range measurement, with five points a_i given. The range measurement ρ_i are the distances of these points to the 'true' point (1,1), plus some measurement errors. We define the residual vector as $f_i(x) = \|x - a_i\| - \rho_i$, with

$i = 1, \dots, 5$, thus $f(x) = \begin{bmatrix} \sqrt{(x_0 - a_{10})^2 + (x_1 - a_{11})^2} - \rho_1 \\ \vdots \\ \sqrt{(x_0 - a_{50})^2 + (x_1 - a_{51})^2} - \rho_5 \end{bmatrix}$. The Jacobian is a 5×2 matrix $\begin{bmatrix} \frac{x_0 - a_{10}}{\|x - a_1\|} & \frac{x_1 - a_{11}}{\|x - a_1\|} \\ \vdots & \vdots \\ \frac{x_0 - a_{50}}{\|x - a_5\|} & \frac{x_0 - a_{51}}{\|x - a_5\|} \end{bmatrix}$. We run the algorithm for three different starting points and note that one of them does not converge near to the true solution.

```
# Five locations ai in a 5 by 2 matrix
A = np.array([[1.8, 2.5], [2.0, 1.7], [1.5, 1.5], [1.5, 2.0], [2.5, 1.5]])
# Vector of measured distances to five locations.
rhos = np.array([1.87288, 1.23950, 0.53672, 1.29273, 1.49353])

dist = lambda x: np.sqrt(np.sum((x-A)**2, axis=1))
f = lambda x: dist(x).reshape(5, 1) - rhos.reshape(5,1)
Df = lambda x: (x-A)/dist(x)[:,None]

p1 = np.linspace(0, 4, 100)
p2 = np.linspace(0, 4, 100)
P1, P2 = np.meshgrid(p1, p2)
p = np.vstack([P1.ravel(), P2.ravel()])
Z = np.array([np.sum(f(p[:,i])**2) for i in range(p.shape[1])]).reshape((100,100))
ax = plt.contour(P1, P2, Z, 20, cmap='RdGy')
ax.axes.grid(True)
x, y = np.where(Z==np.min(Z))
x, y = x[0], y[0]
ax.axes.plot(P1[x,y], P2[x,y], '*', color='k', markersize=20)
for i in A:
    ax.axes.plot(i[0], i[1], 'o', color='k', markersize=5)
plt.colorbar()

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(P1, P2, Z, rstride=3, cstride=3, alpha=0.8, cmap=plt.cm.CMRmap)
plt.tight_layout()
```

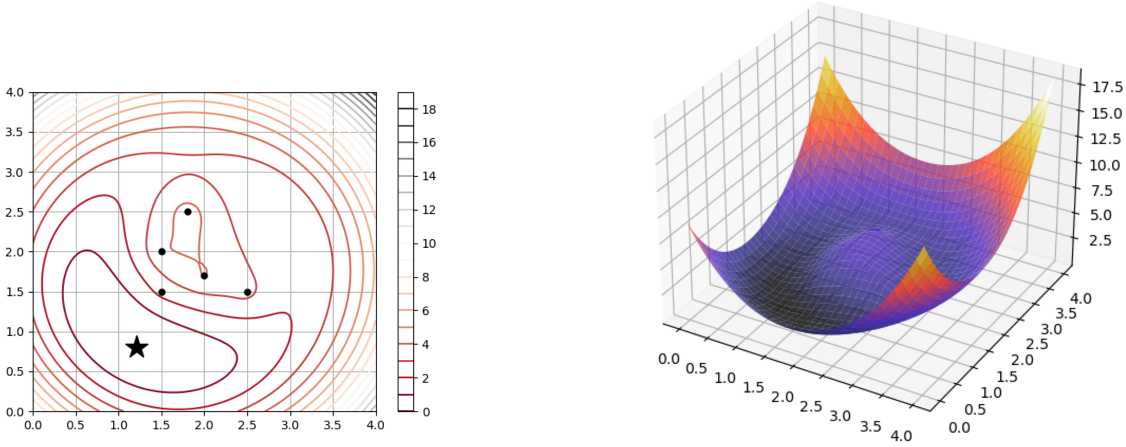

Figure 18: Contour lines of $\|f(x)\|^2$. The dots show the point a_i, and the point marked with a star is the point that minimizes $\|f(x)\|^2$. The right plot shows the same surface in 3d.

```
lam = 0.1
fx, l_list, x_star = {},{},{}
for x0 in np.array([[1.8, 3.5], [3.0, 1.5],[2.2, 3.5]]):
    fx[tuple(x0)], l_list[tuple(x0)], x_star[tuple(x0)] = lm(x0, lam, 10)
pd.DataFrame(fx).plot(style='o-', grid=True)
pd.DataFrame(l_list).plot(style='o-', grid=True)

x_star
```

```
>>
{(1.8, 3.5): array([1.18248556, 0.82422891]),
 (3.0, 1.5): array([1.18248563, 0.82422915]),
 (2.2, 3.5): array([2.98411753, 2.1243895 ])}
```

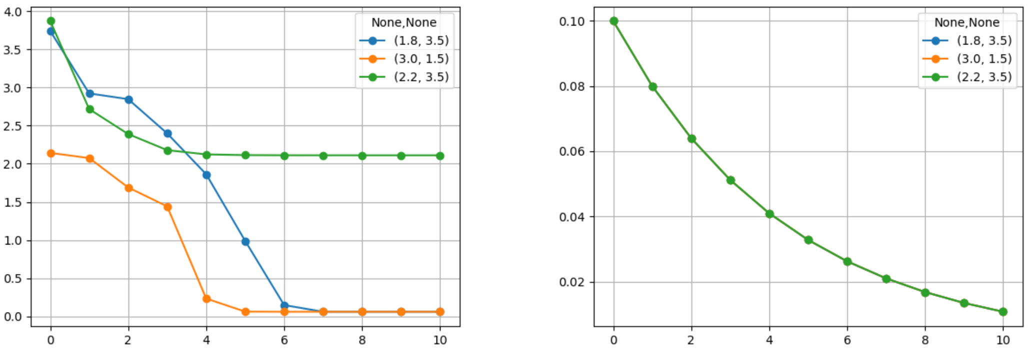

Figure 19: Cost function $\|f(x^{(k)})\|^2$ and trust parameter $\lambda^{(k)}$ versus iteration number k for the three starting points.

□

Example 14. *(Nonlinear model fitting)* In nonlinear model fitting we are given a set of data $x^{(1)}, \dots, x^{(N)}, y^{(1)}, \dots, y$ where the n-vectors $x^{(i)}$ are the feature vectors, and the scalars $y^{(i)}$ are the associated outcomes. We fit a model of the general form $y \approx \hat{f}(x;\theta)$ to the given data, where the p-vector θ contains the model parameters. We choose θ by minimizing $\sum_{i=1}^{N}(\hat{f}(x^{(i)};\theta) - y^{(i)})^2$, along with regularization term.

In this example we fit a exponentially decaying sinusoid $\hat{f}(x;\theta) = \theta_1 e^{\theta_2 x}\cos(\theta_3 x + \theta_4)$, with four parameters, and $N = 60$ data points.

```
# Use these parameters to generate data
theta_x = np.array([1, -0.2, 2*np.pi/5, np.pi/3])
xd = np.vstack([5*np.random.random((30,1)), 5+15*np.random.random((30,1))])
yd = theta_x[0]*np.exp(theta_x[1]*xd)*np.cos(theta_x[2]*xd + theta_x[3])
N = len(yd)
yd = np.multiply(yd.T, (1 + 0.2*np.random.normal(size = N)) +
                0.015*np.random.normal(size = N)).T

ax = pd.Series(yd[:,0], index=xd[:,0]).plot(grid=True, style='o', color='g')

theta0 = np.array([1,0,1,0])
lam = 1

f =  lambda t: t[0]*np.exp(t[1]*xd)*np.cos(t[2]*xd + t[3]) - yd
cf =  lambda t: t[0]*np.exp(t[1]*xd)*np.cos(t[2]*xd + t[3])
sf =  lambda t: t[0]*np.exp(t[1]*xd)*np.sin(t[2]*xd + t[3])
jacobian = lambda t: np.hstack([cf(t)/t[0], xd*cf(t), -xd*sf(t), -sf(t)])
fx_arr, lm_arr, x_star = lm(theta0, lam)

x = np.linspace(0, 20, 500)
t = x_star
y = t[0]*np.exp(t[1]*x)*np.cos(t[2]*x + t[3])
pd.Series(y, index=x).plot(style='-', ax=ax, color='k', grid=True)
x_star
>> array([-0.96257993, -0.22133747,  1.24199956, -2.02368857])
```

We apply the orthogonal distance regression next for data fitting. In ordinary least squares regression we find a curve that minimizes the sum of squares of the vertical errors between the curve and the data points. In *orthogonal distance regression* we use another objective, the sum of the squared distances of N points $(x^{(i)}, y^{(i)})$

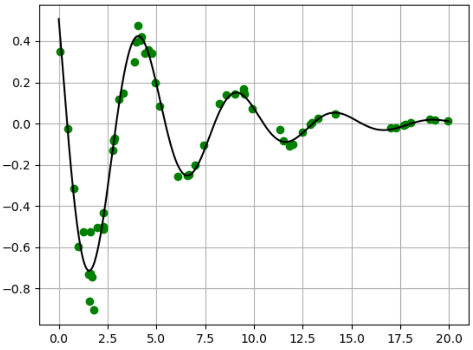

Figure 20: Least squares fit of a function $\hat{f}(x;\theta)$ for $N = 60$ points.

to the graph of $\hat{f}$, i.e. the set of points of the form $(u, \hat{f}(u))$. The model can be found by solving the nonlinear least squares problem

$$\text{minimize } \sum_{i=1}^{N}(\hat{f}(u^{(i)};\theta) - y^{(i)})^2 + \sum_{i=1}^{N} \|u^{(i)} - x^{(i)}\|^2$$

with variables $\theta_1, \dots, \theta_p$ and $u^{(1)}, \dots, u^{(N)}$. This is an example of *error-in-variables model*, since it takes into account errors in the regressors or independent variables. Roughly speaking, we fit a curve that passes near all the data points, as measured by the minimum distance from the data points to the curve.

We fit a cubic polynomial $\hat{f}(x;\theta) = \theta_1 + \theta_2 x + \theta_3 x^2 + \theta_4 x^3$ to $N = 25$ data points. The residual is $\begin{bmatrix} \hat{f}(u^{(1)};\theta) - y^{(1)} \\ \vdots \\ \hat{f}(u^{(N)};\theta) - y^{(N)} \\ u^{(1)} - x^{(1)} \\ \vdots \\ u^{(N)} - x^{(N)} \end{bmatrix}$ and Jacobian is

$$\begin{bmatrix}
1 & u^{(1)} & (u^{(1)})^2 & (u^{(1)})^3 & \theta_2 + 2\theta_3 u^{(1)} + 3\theta_4 (u^{(1)})^2 & 0 & \dots & 0 \\
1 & u^{(2)} & (u^{(2)})^2 & (u^{(2)})^3 & 0 & \theta_2 + 2\theta_3 u^{(2)} + 3\theta_4 (u^{(2)})^2 & \dots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\
1 & u^{(N)} & (u^{(N)})^2 & (u^{(N)})^3 & 0 & 0 & \dots & \theta_2 + 2\theta_3 u^{(N)} + 3\theta_4 (u^{(N)})^2 \\
0 & 0 & 0 & 0 & 1 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & \dots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \dots & 1
\end{bmatrix}.$$

We compare the ordinary least sqaures fit to the orthogonal distance regression. We use OLS solution as the starting point for the LM algorithm. We also show here how to use the *scipy* function instead of our own.

```
# cubic spline fit
from ECONOMETRICS.Boyd.orth_dist_reg_data import orth_dist_reg_data
xd, yd = orth_dist_reg_data()
N = len(xd)
p = 4
theta_ls = ls(np.vander(xd.ravel(), p,increasing=True), yd)
x_in = np.linspace(0.1,1, 100)
y_pred_ls = np.vander(x_in.ravel(), p, increasing=True) @ theta_ls

# ODR
xd, yd = xd.ravel(), yd.ravel()

f = lambda x: np.hstack([np.vander(x[p:], p, increasing=True) @ x[:p] - yd, x[p:]-xd])
jacobian = lambda x: np.block(
    [[np.vander(x[p:], p, increasing=True),
      np.diag((x[1]*x[p:]+2*x[2]*x[p:]+3*x[3]*(x[p:]**2)).ravel())],
     [np.zeros((N, p)), np.eye(N)]])
```

```
x0= np.hstack([theta_ls.ravel(), xd])
from scipy.optimize import least_squares
res = least_squares(f, x0, jacobian, method='lm')
theta_odr = res['x'][:p]
y_pred_odr = np.vander(x_in.ravel(), p, increasing=True) @ theta_odr

ax = pd.Series(yd, index=xd).plot(style='o', color='green', grid=True)
pd.DataFrame({'ls': y_pred_ls[:,0], 'odr': y_pred_odr}, index=x).
            plot(ax=ax, style='-', grid=True)
```

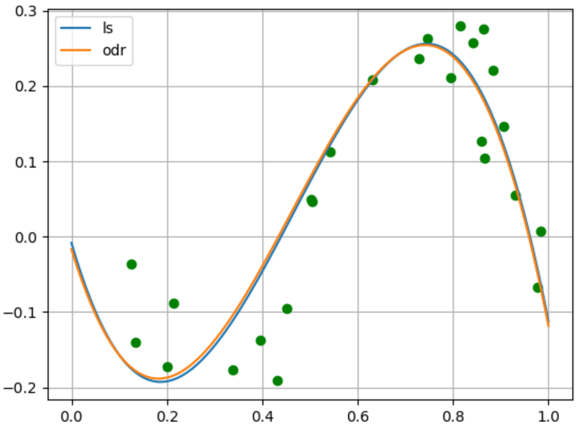

Figure 21: Least squares and orthogonal distance regression fit of a cubic polynomial to 25 data points.

□

We can extend the nonlinear least squares to classification problem. Ideally we would want to minimize $\sum_{i=1}^{N}\left(sign(\tilde{f}(x^{(i)})) - y^{(i)}\right)^2$ for a Boolean classification problem where $f(x) = \theta_1 f_1(x) + \ldots + \theta_p f_p(x)$ to the data points $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, N$, where $y^{(i)} \in \{-1, 1\}$. However, because the *sign* function is not differentiable, Levenberg-Marquardt algorithm would not work on this objective. To resolve this, we replace the sign function with a differentiable *surrogate* the *sigmoid function* $\phi(u) = \frac{e^{2u}-1}{e^{2u}+1}$. We choose θ by solving the nonlinear least squares problem of minimizing $\sum_{i=1}^{N}\left(\phi(\phi(\tilde{f}(x^{(i)}))) - y^{(i)}\right)^2$ using the LM algorithm. We can also add regularization to this objective. This objective can be seen in terms of loss functions $\sum_{i=1}^{N} \ell(\tilde{f}(x^{(i)}), y^{(i)})$ which depends on the continuous prediction $\tilde{f}(x^{(i)})$ and the outcome $y^{(i)}$.

Example 15. *(Handwritten digit classification)* We apply nonlinear least squares classification on the MNIST set of handwritten digits. We first consider the Boolean problem of recognizing the digit zero using the linear features $\tilde{f}(x) = x^T\beta + v$, where x is the 493-vector of pixel intensities. To determine v and β we solve the nonlinear least square problem

$$\text{minimize} \sum_{i=1}^{N}\left(\phi((x^{(i)})^T\beta + v) - y^{(i)}\right)^2 + \lambda\|\beta\|^2,$$

where ϕ is the sigmoid function and λ is a positive regularization parameter, which is different from the trust parameter $\lambda^{(k)}$ in the iterates of the Levenberg-Marquardt algorithm. The residual $(N+m) \times 1$ matrix and Jacobain $(N+m) \times (m+1)$ matrix are

$$f(v,\beta) = \begin{bmatrix} \phi(x^{(1)T}\beta + v) - y^{(1)} \\ \vdots \\ \phi(x^{(N)T}\beta + v) - y^{(N)} \\ \sqrt{\lambda}\beta_1 \\ \vdots \\ \sqrt{\lambda}\beta_m \end{bmatrix}, \; Df = \begin{bmatrix} \phi'(x^{(1)T}\beta + v) & \phi'(x^{(1)T}\beta + v)x_1^{(1)} & \ldots & \phi'(x^{(N)T}\beta + v)x_m^{(N)} \\ \vdots & \vdots & & \vdots \\ \phi'(x^{(N)T}\beta + v) & \phi'(x^{(N)T}\beta + v)x_1^{(N)} & \ldots & \phi'(x^{(N)T}\beta + v)x_m^{(N)} \\ 0 & \sqrt{\lambda} & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \sqrt{\lambda} \end{bmatrix}.$$

We use the least squares solution as the starting value of the parameter vector.

```
############################
# Nonlinear Classification #
############################
import numpy as np, pandas as pd, scipy.linalg as scla, seaborn as sb, matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from keras.datasets import mnist
from sklearn.metrics import confusion_matrix

class classifyMINST:
    def __init__(self, data):
        self.y_train = data[0][1]
        x = data[0][0].reshape(60000, 28 * 28) / 255
        # border removed data
        mask = np.sum(x != 0, axis=0) >= 600
        self.x_train = x[:, mask]
        self.x_test = (data[1][0].reshape(10000, 28 * 28) / 255)[:, mask]
        self.y_test = data[1][1]

    def add_random_features(self, N):
        R = np.random.choice([-1, 1], (N, self.x_train.shape[1]+1))
        xt = np.concatenate([np.ones((60000, 1)), self.x_train], axis=1)
        f = np.clip(xt @ R.T, 0, np.inf)
        self.x_train = np.concatenate([xt, f], axis=1)
        x_val = np.concatenate([np.ones((10000, 1)), self.x_test], axis=1)
        f = np.clip(x_val @ R.T, 0, np.inf)
        self.x_test = np.concatenate([x_val, f], axis=1)

    def info(self):
        print(self.y_train.shape, self.x_train.shape, self.y_test.shape, self.x_test.shape)
        return pd.Series(self.y_train).value_counts()

    def metric(self, truth, prediction):
        return pd.Series({'error rate':np.sum(truth!=prediction)/len(truth),
            'true positive rate': np.sum(truth & prediction)/np.sum(truth),
            'false positive rate': np.sum(~truth & prediction)/np.sum(~truth)}).round(4)*100

    def phi(self, a):
        e2 = np.exp(2 * a)
        return (e2 - 1) / (e2 + 1)

    def dphi(self, a):
        e2 = np.exp(2 * a)
        return 4 * e2 / (e2 + 1) ** 2

    def f(self, beta, reg_lam):
        return np.hstack([self.phi(self.x_train @ beta[1:] + beta[0]) - \
                          (2 * (self.y_train == 0) - 1),
                          np.sqrt(reg_lam) * beta[1:]])

    def Df(self, beta, reg_lam):
        N, m = self.x_train.shape
        d = self.dphi(self.x_train @ beta[1:] + beta[0])[:, None]
        return np.hstack([np.vstack([d, np.zeros((m, 1))]),
                          np.vstack([d * self.x_train, np.sqrt(reg_lam) * np.eye(m)])])

    # RLS
    def rls(self, A, b, lam):
        gm = A.T @ A + np.sqrt(lam) * np.eye(A.shape[1])
        q, r = np.linalg.qr(gm)
        A_inv = scla.solve_triangular(r, q.T, check_finite=False, lower=False)
        return A_inv @ b

    # LM
    def lm(self, x, lam=1, count_max=20, err=1e-4, reg_lam=100):
        fx = self.f(x, reg_lam)
        fxn = np.linalg.norm(fx)
        fxn_arr = [fxn]
        lam_arr = [lam]
        x_arr = [x]
        while (count_max > 0):
            if (fxn_arr[-1] < err): break
            j = self.Df(x, reg_lam)
            DfTf = j.T @ fx[:, None]
            if np.linalg.norm(2 * DfTf) < err: break
            _x = x - self.rls(j, DfTf, lam)[:, 0]
            _fx = self.f(_x, reg_lam)
            _fxn = np.linalg.norm(_fx)
            if _fxn < fxn_arr[-1]:
                lam = 0.8 * lam
                x = _x
                fx = _fx
                fxn = _fxn
            else:
                lam = 2 * lam
            fxn_arr.append(fxn)
            lam_arr.append(lam)
            x_arr.append(x)
            count_max -= 1
        return fxn_arr, lam_arr, x_arr

    def fitls(self):
        ls = LinearRegression(fit_intercept=True).fit(self.x_train, 2 * (self.y_train == 0) - 1)
        return LSResults(ls, self)

    def fit(self, x0=None, lm_lam=1, max_iters=10, err=1e-4, reg_lam=100):
        if x0 is None:
            x0 = self.fitls().betas()
        result = self.lm(x0, lm_lam, max_iters, err, reg_lam)
        return LMResults(result, self)


class LSResults:
    def __init__(self, result, lmobj):
        self.result = result
        self.lmobj = lmobj

    def betas(self):
        return np.hstack([self.result.intercept_, self.result.coef_])

    def predict(self, x):
        return self.result.predict(x)

    def metric(self, x, y):
        return self.lmobj.metric(y == 0, self.predict(x) > 0)


class LMResults:
    def __init__(self, result, lmobj):
        self.results = result
        self.lmobj = lmobj

    def betas(self, i=-1):
        return self.results[2][i]

    def predict(self, x, i=-1):
        beta = self.results[2][i]
        return self.lmobj.phi(x @ beta[1:] + beta[0])

    def metric(self, x, y, i=-1):
        return self.lmobj.metric(y == 0, self.predict(x, i) > 0)
    def confusion_matrix(self, x, y, i=-1):
        return confusion_matrix(y == 0, self.predict(x, i) > 0)

cm = classifyMINST(mnist.load_data())
ls = cm.fitls()
ls.metric(cm.x_train, cm.y_train)
error rate              1.55
true positive rate     87.08
false positive rate     0.31

x0 = ls.betas()
# run for various lambdas, train-test ============
er = {}
for reg_lam in range(-9,6,1):
    print(reg_lam)
    res = cm.fit(x0, lm_lam=1, max_iters=5, reg_lam=10**reg_lam)
    er[10**reg_lam] = pd.Series({
        'train':res.metric(cm.x_train, cm.y_train)['error rate'],
        'test': res.metric(cm.x_test, cm.y_test)['error rate']})
pd.DataFrame(er).T.plot(style='o-', grid=True, logx=True)
```

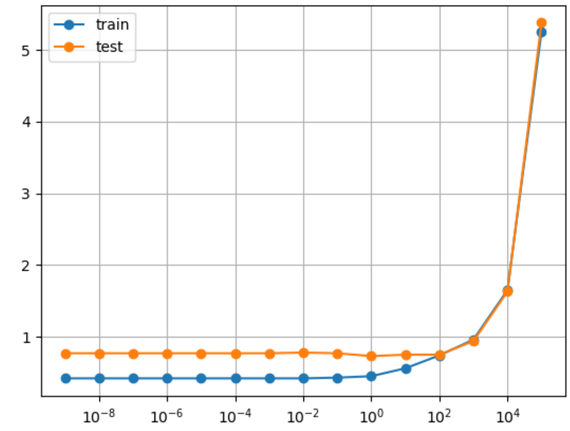

Figure 22: Boolean classification error in percent versus λ.

```
#run for lam=100 ==================================
res = cm.fit(reg_lam=100)
er = {}
for i, beta in enumerate(res.results[2]):
    er[i] = pd.Series({
        'train': res.metric(cm.x_train, cm.y_train, i)['error rate'],
        'test': res.metric(cm.x_test, cm.y_test, i)['error rate']})
pd.DataFrame(er).T.plot(style='o-', grid=True)


res.confusion_matrix(cm.x_train, cm.y_train)
res.confusion_matrix(cm.x_test, cm.y_test)

beta = res.betas()
yh = cm.x_train @ beta[1:] + beta[0]
ax = pd.Series(yh[cm.y_train==0]).hist(bins=100, color='blue', density=True, alpha=0.6)
pd.Series(yh[cm.y_train!=0]).hist(ax=ax, bins=100, color='red', density=True, alpha=0.6)
ax.axvline(x=0, color='k')
```

Figure 22 shows the classification error on the training and test sets as a function of the regularization parameter λ. For $\lambda = 100$, the classification and training error is about 0.7%, less than half the 1.55% error of the Boolean least squares classifier that used the same features. This improvement in performance, by more than a factor

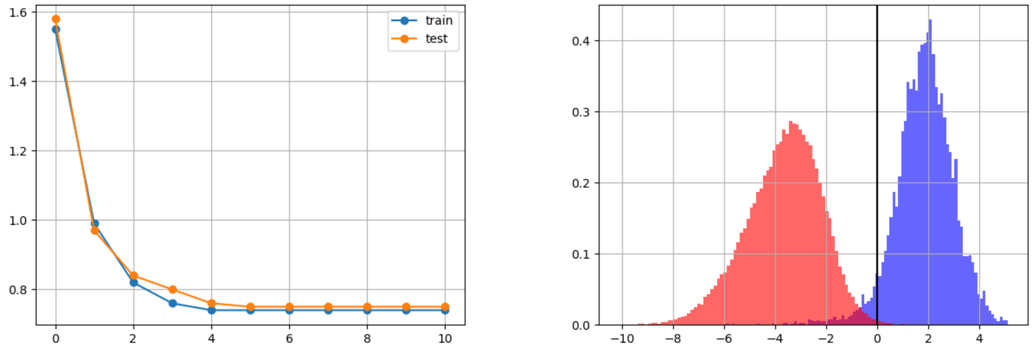

Figure 23: Training and test error versus Levenberg-Marquardt iteration for $\lambda = 100$. The distribution of the values of $\tilde{f}(x^{(i)})$ used in the Boolean classifier for recognizing digit zero.

of two, *comes from minimizing an objective that is closer to what we want*, i.e. the number of prediction errors on the training set, than the surrogate linear least squares objective. The confusion matrix for train and test sets, shown in the code, also verify the good performance. Figure 23 shows the distribution of the values of $\tilde{f}(x^{(i)})$ for the two classes of the data set. The figure on the left also shows that *though Levenberg-Marquardt algorithm may take tens of iterations to converge, but the classification error minimization is reached within first few iterations!*. This phenomenon is due to *maximization of the margin* even after the classification error has stopped reducing.

We now turn to the black magic of feature engineering once again, in the nonlinear context. After adding 5000 random features as used before we show the training and test classification error once again in fig.24.

```
cm.add_random_features(5000)
ls = cm.fitls()
ls.metric(cm.x_train, cm.y_train)
# error rate               0.22
# true positive rate      98.18
# false positive rate      0.04

# run for various lambdas, train-test
x0 = np.hstack([x0, np.zeros(cm.x_train.shape[1]-len(x0)+1)])
er = {}
for reg_lam in range(-2,7,1):
    print(reg_lam)
    res = cm.fit(x0, lm_lam=1, max_iters=5, reg_lam=10**reg_lam)
    er[10**reg_lam] = pd.Series({
        'train':res.metric(cm.x_train, cm.y_train)['error rate'],
        'test': res.metric(cm.x_test, cm.y_test)['error rate']})
pd.DataFrame(er).T.plot(style='o-', grid=True, logx=True)
```

The error on the training set is zero for small λ. For $\lambda = 100$, the error on the test set is 0.25%, with the confusion matrix shown in the code. The distribution of $\tilde{f}(x^{(i)})$ on the training set is shown in fig. 25 showing perfect separation. Figure 25 also shows the classification error as the iterations proceed for = 100 and shows that we essentially need just 1 or 2 steps to get very near the 'optimal' solution.

```
#run for lam=100 =================================
res = cm.fit(x0, max_iters=10, reg_lam=100)
_er = {}
for i, beta in enumerate(res.results[2]):
    _er[i] = pd.Series({
        'train': res.metric(cm.x_train, cm.y_train, i)['error rate'],
        'test': res.metric(cm.x_test, cm.y_test, i)['error rate']})
pd.DataFrame(_er).T.plot(style='o-', grid=True)
res.metric(cm.x_test, cm.y_test)
# error rate               0.25
# true positive rate      98.37
# false positive rate      0.10
```

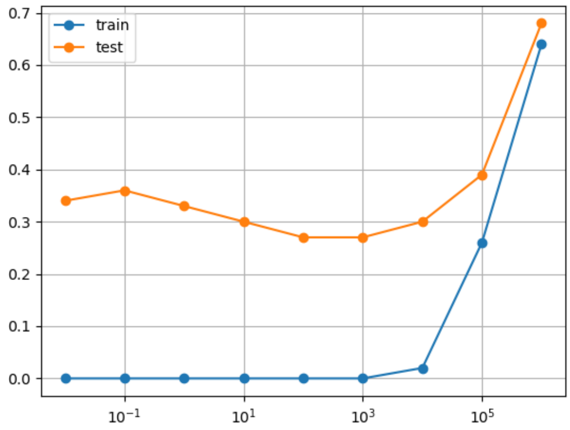

Figure 24: Boolean classification error in percent versus λ after adding 5000 random features.

```
res.confusion_matrix(cm.x_train, cm.y_train)
# array([[54077,      0],
#        [    0,   5923]])
res.confusion_matrix(cm.x_test, cm.y_test)
# array([[9011,    9],
#        [  16,   964]])

beta = res.betas()
yh = cm.x_train @ beta[1:] + beta[0]
ax = pd.Series(yh[cm.y_train==0]).hist(bins=100, color='blue', density=True, alpha=0.6)
pd.Series(yh[cm.y_train!=0]).hist(ax=ax, bins=100, color='red', density=True, alpha=0.6)
ax.axvline(x=0, color='k')
```

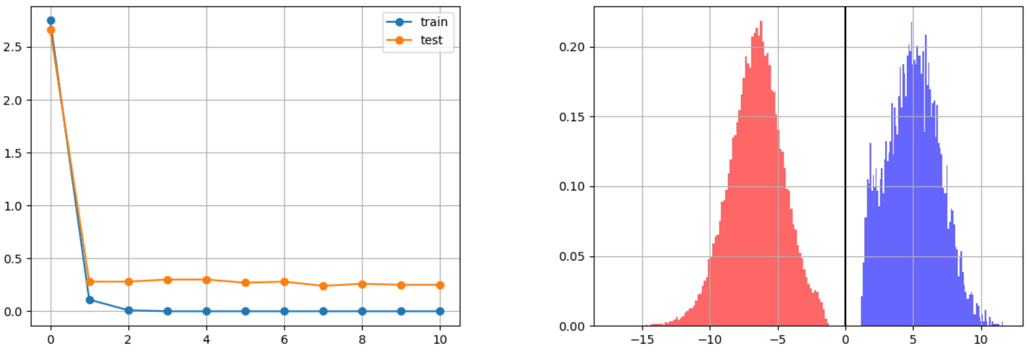

Figure 25: Training and test errors vesurs Levenberg-Marquardt iteration for $\lambda = 100$. The distribution of the values of $\tilde{f}(x^{(i)})$ used in the Boolean classifier for recognizing the digit zero, after addition of 5000 new features.

We now apply the nonlinear least squares method to the multi-class classification of recognizing the ten digits in the MINST data set. For each digit k we compute a Boolean classifier $\tilde{f}_k(x) = x^T\beta_k + v_k$ by solving a regularized nonlinear least squares problem. The same value of λ is used in the ten nonlinear least squares problem The Boolean classifiers are combined into a multi-class classifier $\hat{f}(x) = \underset{k=1,\ldots,10}{\operatorname{argmax}}(x^T\beta_k + v_k)$.

```
# additional functions added to take care of multi-group case
# classifyMINST
def mcmetric(self, truth, prediction):
    sr = pd.Series({'error rate': np.sum(truth != prediction) / len(truth)})
    for i in np.unique(truth):
        imask = truth == i
        sr['true %s rate' % (i)] = np.sum(prediction[imask] == i) / np.sum(imask)
    return sr.round(4) * 100
```

```
def fitls_multi(self):
    ohe = 2 * OneHotEncoder().fit(cm.y_train.reshape(-1, 1)).
            transform(cm.y_train.reshape(-1, 1)).toarray() - 1
    ls = LinearRegression(fit_intercept=True).fit(self.x_train,ohe)
    return LSResults(ls, self)

def fit_multi(self, x0=None, lm_lam=1, max_iters=10, err=1e-4, reg_lam=100):
    if x0 is None:
        x0 = self.fitls_multi().betas()
    result = {}
    for group in range(10):
        print(group)
        self.group = group
        result[group] = self.lm(x0[group], lm_lam, max_iters, err, reg_lam)
    return LMResults(result, self, True)

# LMResults
def predict(self, x, i=-1):
    if self.multi:
        pred = []
        for j in range(10):
            beta = self.results[j][2][i]
            pred.append(self.lmobj.phi(x @ beta[1:] + beta[0]))
        return np.vstack(pred).T
    else:
        beta = self.results[2][i]
        return self.lmobj.phi(x @ beta[1:] + beta[0])
def metric_multi(self, x, y, i=-1):
    return self.lmobj.mcmetric(y, np.argmax(self.predict(x, i), 1))

# least square base case
cm = classifyMINST(mnist.load_data())
res = cm.fitls_multi()
er = pd.DataFrame({
    'train': res.metric_multi(cm.x_train, cm.y_train),
    'test': res.metric_multi(cm.x_test, cm.y_test)
})
#               train   test
# error rate    14.45   13.93
# true 0 rate   95.71   96.33
# true 1 rate   97.05   97.53
# true 2 rate   79.84   78.97
# true 3 rate   84.00   87.52
# true 4 rate   88.82   89.92
# true 5 rate   73.31   73.54
# true 6 rate   92.06   91.44
# true 7 rate   86.88   85.89
# true 8 rate   75.49   77.62
# true 9 rate   79.71   79.58
x0 = res.betas()
er = {}
for reg_lam in range(-5,4,1):
    print(reg_lam)
    res = cm.fit_multi(x0, lm_lam=1, max_iters=5, reg_lam=10**reg_lam)
    er[10**reg_lam] = pd.Series({
        'train':res.metric_multi(cm.x_train, cm.y_train)['error rate'],
        'test': res.metric_multi(cm.x_test, cm.y_test)['error rate']})
pd.DataFrame(er).T.plot(style='o-', grid=True, logx=True)
```

Figure 26 shows the classification error versus λ. For $\lambda = 0.1$ fig. 27 shows that the classification error settles down by 6th iteration to 7.6% which is half of the least square error of 13.93%.

```
res = cm.fit_multi(reg_lam=0.1)
_er = {}
for i in range(10):
    _er[i] = pd.Series({
```

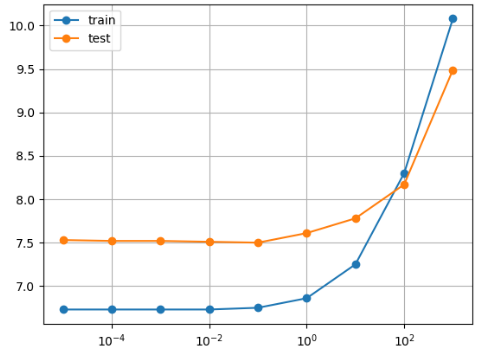

Figure 26: Multiclass classification error in percent versus λ.

```
        'train':res.metric_multi(cm.x_train, cm.y_train, i)['error rate'],
        'test': res.metric_multi(cm.x_test, cm.y_test, i)['error rate']})
pd.DataFrame(_er).T.plot(style='o-', grid=True)

confusion_matrix(cm.y_train, np.argmax(res.predict(cm.x_train), 1))
array([[5803,    1,    5,    7,    4,   14,   23,    9,   53,    4],
       [   1, 6598,   34,   19,    5,   21,    4,   15,   37,    8],
       [  29,   24, 5499,   63,   54,   25,   58,   57,  133,   16],
       [  12,   10,  132, 5573,   11,  118,   25,   44,  140,   66],
       [  16,   22,   17,    5, 5502,    7,   33,    5,   73,  162],
       [  37,    6,   35,  120,   37, 4907,   78,   21,  130,   50],
       [  27,   13,   16,    2,   22,   97, 5691,    2,   47,    1],
       [  13,   15,   63,   21,   53,   13,    3, 5924,   27,  133],
       [  43,   74,   47,  113,   40,  155,   44,   22, 5255,   58],
       [  42,    8,   11,   89,  153,   52,    3,  140,  107, 5344]])

confusion_matrix(cm.y_test, np.argmax(res.predict(cm.x_test), 1))
array([[ 959,    0,    1,    1,    0,    3,    5,    3,    7,    1],
       [   0, 1112,    3,    2,    0,    2,    4,    1,   10,    1],
       [   4,    5,  935,   13,    6,    4,   10,    9,   42,    4],
       [   4,    0,   19,  921,    1,   17,    2,   10,   25,   11],
       [   1,    3,    4,    3,  915,    0,    9,    2,   12,   33],
       [   7,    2,    3,   31,   11,  783,   18,    7,   22,    8],
       [  10,    3,    6,    0,    5,   21,  904,    1,    8,    0],
       [   1,    6,   23,    7,    9,    4,    0,  947,    3,   28],
       [  11,   12,    6,   20,   12,   24,    8,   11,  867,    3],
       [   7,    6,    0,   12,   33,    9,    1,   17,   26,  898]])
```

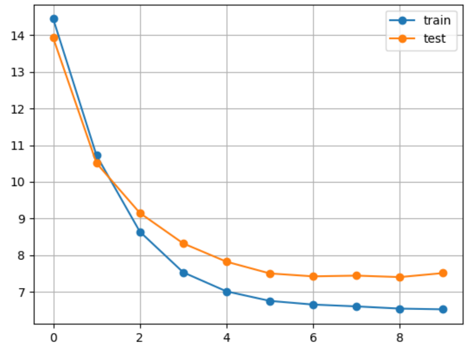

Figure 27: Classification error for test and train set versus Levenberg-Marquardt iteration for $\lambda = 0.1$.

Finally, we get down to adding 5000 random features.

```
cm.add_random_features(5000)
res = cm.fitls_multi()
```

```
er = pd.DataFrame({
    'train': res.metric_multi(cm.x_train, cm.y_train),
    'test': res.metric_multi(cm.x_test, cm.y_test)
})
#                train   test
# error rate      1.55   2.70
# true 0 rate   99.29   98.88
# true 1 rate   99.08   99.03
# true 2 rate   98.41   96.51
# true 3 rate   97.77   97.33
# true 4 rate   98.61   97.45
# true 5 rate   98.36   97.76
# true 6 rate   99.21   98.23
# true 7 rate   98.12   95.91
# true 8 rate   98.33   97.02
# true 9 rate   97.31   94.85

# run for various lambdas, train-test
x0 = np.hstack([x0, np.zeros((x0.shape[0], cm.x_train.shape[1] - x0.shape[1] + 1))])
er = {}
for reg_lam in range(-2,6,1):
    print(reg_lam)
    res = cm.fit_multi(x0, lm_lam=1, max_iters=5, reg_lam=10**reg_lam)
    er[10**reg_lam] = pd.Series({
        'train':res.metric_multi(cm.x_train, cm.y_train)['error rate'],
        'test': res.metric_multi(cm.x_test, cm.y_test)['error rate']})
pd.DataFrame(er).T.plot(style='o-', grid=True, logx=True)
```

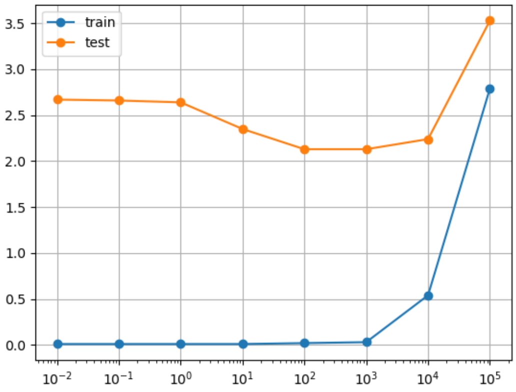

Figure 28: Multiclass classification error in percent versus λ after adding 5000 random features.

Figure 28 shows the error rates when we add the 5000 randomly generated features. The traning and test error rates are now 0.02% and 2.14%. The test set confusion matrix for $\lambda = 100$ is shown in the code output. The classifier has matched human performance in classifying the digits correctly. Further sophisticated feature engineering can bring the test error well below what humans can achieve.

```
# run for lam=100 =================================
res = cm.fit(x0, max_iters=10, reg_lam=100)
_er = {}
for i, beta in enumerate(res.results[2]):
    _er[i] = pd.Series({
        'train': res.metric(cm.x_train, cm.y_train, i)['error rate'],
        'test': res.metric(cm.x_test, cm.y_test, i)['error rate']})
pd.DataFrame(_er).T.plot(style='o-', grid=True)

confusion_matrix(cm.y_train, np.argmax(res.predict(cm.x_train), 1))
array([[5923,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0, 6742,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    1, 5957,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0, 6129,    1,    0,    0,    0,    1,    0],
       [   0,    1,    0,    0, 5840,    0,    0,    1,    0,    0],
       [   0,    0,    0,    0,    0, 5420,    1,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0, 5918,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0, 6265,    0,    0],
       [   0,    0,    0,    0,    0,    0,    1,    0, 5850,    0],
       [   0,    0,    1,    0,    1,    0,    0,    0,    0, 5947]])
confusion_matrix(cm.y_test, np.argmax(res.predict(cm.x_test), 1))
array([[ 968,    1,    2,    1,    0,    3,    1,    2,    1,    1],
       [   0, 1126,    2,    1,    0,    0,    3,    1,    2,    0],
       [   3,    0, 1008,    4,    2,    2,    1,    5,    6,    1],
       [   0,    0,    5,  983,    0,    9,    1,    3,    4,    5],
       [   1,    1,    4,    0,  960,    0,    2,    1,    2,   11],
       [   3,    0,    2,    8,    1,  867,    4,    1,    5,    1],
       [   3,    2,    0,    0,    3,    6,  944,    0,    0,    0],
       [   0,    4,    7,    2,    1,    0,    0, 1003,    4,    7],
       [   4,    0,    4,   10,    3,    4,    4,    1,  939,    5],
       [   3,    3,    3,    6,    8,    5,    1,    4,    3,  973]])
```

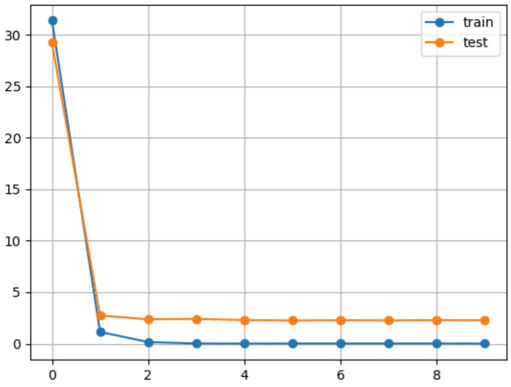

Figure 29: Classification error for the test and train set versus Levenberg-Marquardt iteration for $\lambda = 100$ with 5000 random features added.

□

7 Constrained nonlinear least squares

We extend the nonlinear least squares problem that includes nonlinear constraints.

$$\begin{aligned}&\text{minimize } \|f(x)\|^2\\&\text{subject to } g(x) = 0,\end{aligned}$$

where x is the n-vector to be found. $f(x)$ is a m-vector, and $g(x)$ is a p-vector. $f_i(x)$, $i = 1, \dots, m$ is the $i$the residual and $g_j(x) = 0$, $j = 1, \dots, p$ is the jth constraint. We can leverage Levenberg-Marquardt algorithm for solving this problem. When g is afine we can simply add the equation to the set of residuals and solve it as before, the real challenge is when g is nonlinear. The Lagrangian of the problem is $\mathcal{L}(x, z) = \|\underset{m\times 1}{f(x)}\|^2 + \underset{p\times 1}{g(x)}^T \underset{p\times 1}{z}$. The optimality condition can be derived to be

$$2Df(\hat{x})^T f(\hat{x}) + Dg(\hat{x})^T \hat{z} = 0, \qquad g(\hat{x}) = 0.$$

These conditions are not sufficient, but necessary. For this reason the algorithms are heuristic in nature.

7.1 Penalty algorithm

We can think of the equality constrained problem to be a limits of a bi-objective problem with objectives $\|g(x)\|^2$ and $\|g(x)\|^2$, as the weight on the second objective increases to infinity, $\|f(x)\|^2 + \mu\|g(x)\|^2$. This can be approximately minimized using the Levenberg-Marquardt algorithm applied to $\| \begin{bmatrix} f(x) \\ \sqrt{\mu} g(x) \end{bmatrix} \|^2$. If we solve this for large enough μ, the second term is a *penalty*. Thus, the penalty algorithm for constrained nonlinear least sqaures goes as follows. For $k = 1, 2, \ldots, k^{max}$

- Solve the unconstrained nonlinear least squares problem by setting $x^{(k+1)}$ to the approximate minimizer of $\|f(x)\|^2 + \mu^{(k)}\|g(x)\|^2$ using the Levenberg-Marquardt algorithm, starting from the initial point $x^{(k)}$.
- Update $\mu^{(k+1)} = 2\mu^{(k)}$.

The penalty algorithm is stopped early if $\|g(x^{(k)})\|$ is small enough, i.e. the equality constraint is almost satisfied. For very high values of μ, the Levenberg-Marquardt algorithm can take too many iterations or simply fail. The augmented Lagrangian algorithm gets around this drawback. We can define $\hat{z}^{(k+1)} = 2\mu^{(k)} g(x^{(k+1)})$ as our estimate of a suitable Lagrange multiplier in iteration $k + 1$.

7.2 Augmented Lagrangian algorithm

The proposal called *Hestenes-Powell algorithm* addresses the difficulty associated with the penalty parameter $\mu^{(k)}$ becoming very large. The Augmented Lagrangian problem is

$$\begin{aligned} &\text{minimize } \|f(x)\|^2 + \mu\|g(x)\|^2 \\ &\text{subject to } g(x) = 0. \end{aligned}$$

with the Lagrangian with parameter $\mu > 0$, define by $\mathcal{L}_\mu(x, z) = \|f(x)\|^2 + g(x)^T z + \mu\|g(x)\|^2$. This problem is equivalent to the original constrained nonlinear least squares problem. We note that this Lagrangian is same as $\|f(x)\|^2 + \mu\|g(x) + \frac{z}{2\mu}\|^2 - \mu\|\frac{z}{2\mu}\|^2$, where the last term is independent of the variable of interest x and hence does not effect the choice of x. Thus we minimize $\|f(x)\|^2 + \mu\|g(x) + \frac{z}{2\mu}\|^2$ expressed as

$$\| \begin{bmatrix} f(x) \\ \sqrt{\mu} g(x) + \frac{z}{2\sqrt{\mu}} \end{bmatrix} \|^2.$$

This can be approximately minimized using the Levenberg-Marquardt algorithm. Any minimizer of $\mathcal{L}_\mu$ also satisfies the optimality condition. It also suggest an update for z as $\tilde{z} = z + 2\mu g(\tilde{x})$. The algorithm alternates between minimizing the augmented Lagrangian and updating the parameter z. The penalty parameter μ increased only when needed, when $\|g(x)\|$ does not sufficiently decrease. For a given differentiable function $f : \boldsymbol{R}^n \to \boldsymbol{R}^m$ and $g : \boldsymbol{R}^n \to \boldsymbol{R}^p$, and an initial point $x^{(1)}$. Set $x^{(1)} = 0$, $\mu^{(1)} = 1$. For $k = 1, 2, \ldots, k^{max}$

- Solve unconstrained nonlinear least sqaures problem. Set $x^{(k+1)}$ to be the approximate minimizer of $\|f(x)\|^2 + \mu\|g(x) + \frac{z}{2\mu}\|^2$ using Levenberg-Marquardt algorithm, starting form initial point $x^{(k)}$.
- Update $z^{(k+1)} = z^{(k)} + 2\mu^{(k)} g(x^{(k+1)})$.
- Update $\mu^{(k+1)} = \begin{cases} \mu^{(k)} & \|g(x^{(k+1)})\| < 0.25\|g(x^{(k)})\| \\ 2\mu^{(k)} & \|g(x^{(k+1)})\| \geq 0.25\|g(x^{(k)})\| \end{cases}$.

The algorithm stops if $g(x^{(k)})$ is very small. This algorithm works much better in practice because the penalty parameter $\mu^{(k)}$ does not need to increase as much as the algorithm proceeds.

Example 16. *(Non linear residual with constraints)* We are given $f(x_1, x_2) = \begin{bmatrix} x_1 + e^{-x_2} \\ x_1^2 + 2x_2 + 1 \end{bmatrix}$, $g(x_1, x_2) = x_1 + x_1^3 + x_2 + x_2^2$. Figure 30 shows the contour lines of the cost function $\|f(x)\|^2$ and the constraint $g(x)$.

```
def f(x):
    return np.array([[x[0]+np.exp(-x[1]), x[0]**2+2*x[1]+1]]).T
def g(x):
    return x[0] + x[0]**3 + x[1] + x[1]**2

# f(x)
p1 = np.linspace(-1, 1, 100)
```

```
p2 = np.linspace(-1, 1, 100)
P1, P2 = np.meshgrid(p1, p2)
p = np.vstack([P1.ravel(), P2.ravel()])
Z = np.sum(f(p)**2, axis=1).reshape((100,100))
ax = plt.contour(P1, P2, Z, 8, colors='black')
x, y = np.where(Z==np.min(Z))
x, y = x[0], y[0]
ax.axes.plot(P1[x,y], P2[x,y], '*', color='k', markersize='20')
plt.clabel(ax, inline=True, fontsize=10)
# g(x)
p1 = np.linspace(-1, 1, 100)
p2 = np.linspace(-1, 1, 100)
P1, P2 = np.meshgrid(p1, p2)
p = np.vstack([P1.ravel(), P2.ravel()])
Z = (g(p)).reshape((100,100))
ax = plt.contour(P1, P2, Z, 5, colors='red')
plt.clabel(ax, inline=True, fontsize=10)
ax.axes.grid(True)
```

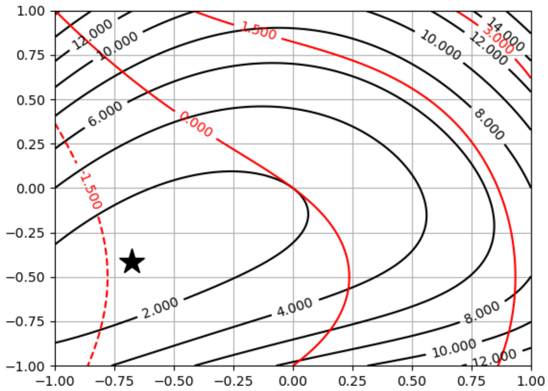

Figure 30: Contour lines of the cost function $\|f(x)\|^2$ in black and the constraint function $g(x)$ in red for a nonlinear least squares problem in two variables with one equality constraint.

The print $\hat{x} = (0, 0)$ is the optimal solution corresponding to the Lagrange multiplier $\hat{z} = -2$. One can verify that $g(\hat{x}) = 0$ and $2Df(\hat{x})^T f(\hat{x}) + Dg(\hat{x})^T \hat{z} = 0$. The star at $x = (-0.666, -0.407)$ indicates the position of the unconstrained minimizer of $\|f(x)\|^2$.

```
def Df(x):
    return np.array([[1, -np.exp(-x[1])],[2*x[0], 2]])
def Dg(x):
    return np.array([1+3*x[0]**2, 1+2*x[1]])
def penalty_algo(f, Df, g, Dg, x, mu=1, iter_max=100, tol=1e-4):
    _f = lambda _x: np.hstack([f(_x), np.sqrt(mu)*g(_x)])
    _Df = lambda _x: np.vstack([Df(_x), np.sqrt(mu)*Dg(_x)])
    _o = lambda _x: np.linalg.norm(2*Df(_x).T @ f(_x) + 2 * mu * Dg(_x).T * g(_x))
    res_f = [np.linalg.norm(g(x))]
    res_o = [_o(x)]
    res_i = [0]
    mu_i = [mu]
    while iter_max>0:
        _lm_res = lm(_f, _Df, x)
        x = _lm_res[2][-1]
        res_f.append(np.linalg.norm(g(x)))
        res_o.append(_o(x))
        res_i.append(len(_lm_res[0]))
        mu_i.append(mu)
        if res_f[-1] < tol: break
        mu = 2*mu
        iter_max -= 1
    return x, res_f, res_o, res_i, mu_i

x_pa, rf, ro, ri_pa, mi_pa = penalty_algo(f, Df, g, Dg, np.array([0.5,-0.5]))
pd.DataFrame({'Feasibility': rf, 'Optimality': ro}, index=np.cumsum(ri_pa)).\
    plot(grid=True, style='o-', drawstyle="steps-post", logy=True)
print(x_pa)
>> [-3.31855595e-05 -2.78464833e-05]
```

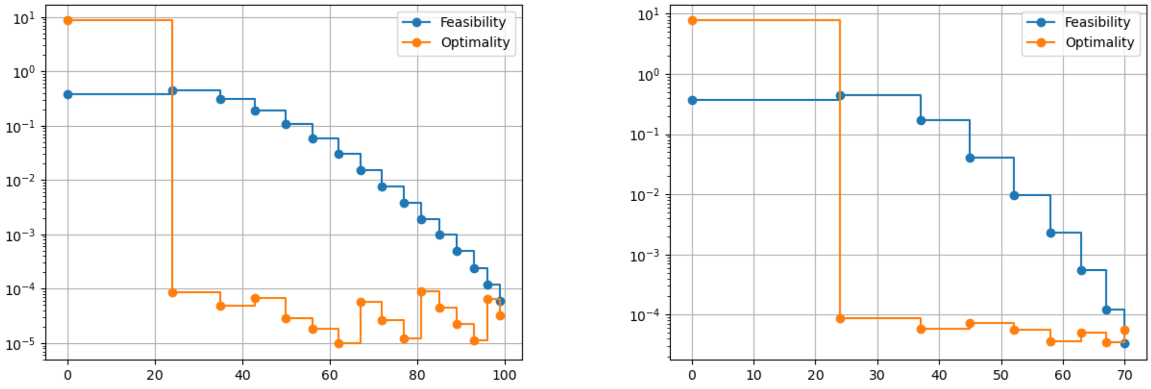

Figure 31: Feasibility and optimality condition errors versus the cumulative number of Levenberg-Marquardt iterations in the penalty (left) and augmented Lagrangian algorithm (right).

We first show the penalty algorithm convergence in fig. 31. The two lines show the absolute value of the feasibility residual $|g(x^{(k)})|$ and the norm of the optimality condition residual, $\|2Df(x^{(k)})^T f(x^{(k)}) + Dg(x^{(k)})^T z^{(k)}\|$. The vertical jump in the optimality condition norm occurs in steps 2 and 3 of the augmented Lagrangian algorithm as well as penalty algorithm. Figure 32 shows the value of the penalty parameter μ versus the cumulative number of Levenberg-Marquardt iterations in the two algorithms.

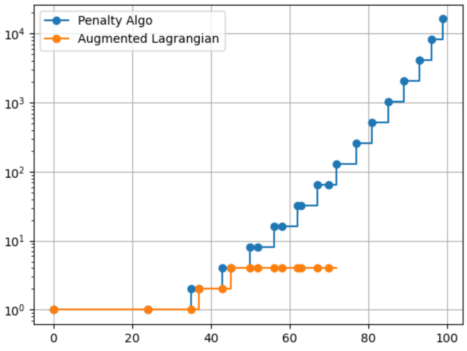

Figure 32: Penalty parameter μ versus cumulative number of Levenberg-Marquardt iterations in the augmented Lagrangian algorithm and the penalty algorithm.

```
def augmented_lagrangian_algo(f, Df, g, Dg, x, mu=1, iter_max=100, tol=1e-4):
    z = np.zeros(len(g(x)))
    _f = lambda _x: np.hstack([f(_x), np.sqrt(mu) * g(_x) + z/2/np.sqrt(mu)])
    _Df = lambda _x: np.vstack([Df(_x), np.sqrt(mu) * Dg(_x)])
    _o = lambda _x: np.linalg.norm(2 * Df(_x).T @ f(_x) + Dg(_x).T * z)
    res_f = [np.linalg.norm(g(x))]
    res_o = [_o(x)]
    res_i = [0]
    mu_i = [mu]
    while iter_max > 0:
        _lm_res = lm(_f, _Df, x)
        x = _lm_res[2][-1]
        print(x)
        z = z + 2 * mu * g(x)
        res_f.append(np.linalg.norm(g(x)))
        res_o.append(_o(x))
        res_i.append(len(_lm_res[0]))
        mu_i.append(mu)
        if res_f[-1] < tol: break
        if res_f[-1] >= 0.25 * res_f[-2]:
            mu = 2 * mu
        iter_max -= 1
    return x, res_f, res_o, res_i, mu_i, z

x_al, rf, ro, ri_al, mi_al, z = augmented_lagrangian_algo(f, Df, g, Dg, np.array([0.5,-0.5]))
pd.DataFrame({'Feasibility': rf, 'Optimality': ro}, index=np.cumsum(ri_al)).\
    plot(grid=True, style='o-', drawstyle="steps-post", logy=True)
print(x_al, z)

mu = pd.DataFrame({'Penalty Algo': pd.Series(mi_pa, index=np.cumsum(ri_pa)),
                'Augmented Lagrangian': pd.Series(mi_al, index=np.cumsum(ri_al))})
t = mu.index[mu.index>mu['Augmented Lagrangian'].last_valid_index()]
mu = mu.ffill()
mu['Augmented Lagrangian'][t]=np.nan
mu.plot(grid=True, style='o-', drawstyle="steps-post", logy=True)
>> [-1.79855232e-05 -1.49904382e-05] [-1.99995875]
```

□

A nonlinear Dynamical system has the form of an iteration $x_{k+1} = f(x_k, u_k)$, $k = 1, 2, \ldots, N$, where n-vector x_k is the state, and the m-vector u_k is the input or control, at the time period k. The function $f : \boldsymbol{R}^{n+m} \to \boldsymbol{R}^n$ specifies what the next state is. When f is an affine function this reduces to linear dynamical system. The goal is to choose $u1, \ldots, u_{N-1}$ to achieve some goal for the state and input trajectories. In many problems initial state x_1 and final state x_N are given. Subject to these constraints, we may wish the control inputs to be small and smooth, which suggests that we minimize $\sum\limits_{k=1}^{N} \|u_k\|^2 + \gamma \sum\limits_{k=1}^{N-1} \|u_{k+1} - u_k\|^2$, where $\gamma > 0$ is a parameter used to trade off input size and smoothness. In many nonlinear control problems the objective also involves the state trajectory.

Example 17. *(Control of a car)* Consider a car with the back wheel axis position $p = (p_1, p_2)$ and orientation angle θ with the horizontal axis. The car has a known wheelbase of length L, steering angle ϕ, and the speed s. The parameter vector is $(p_1, p_2, \theta, \phi, s)$ which is a function of time. The dynamics of the car is given by $\frac{d}{dt}p_1(t) = s(t)\cos\theta(t)$, $\frac{d}{dt}p_2(t) = s(t)\sin\theta(t)$, and $\frac{d}{dt}\theta(t) = \frac{s(t)}{L}\tan\phi(t)$, with $-\frac{\pi}{2} < \phi(t) < \frac{\pi}{2}$. For a fixed steering angle and speed the car moves in a circle. We can control the speed s and the steering angle ϕ with the goal of moving the car from some given initial position and orientation to a specified final position and orientation over some time period.

We now discretize the equations in time. We take a small time interval steps $t = hk, k = 1, 2, \ldots$, and obtain the approximations to derive nonlinear state equations for the motion with states $x_t = (p_{1t}, p_{2t}, \theta_t)$ and inputs $u_t = (s_t, \phi_t)$ giving the dynamics $x_{t+1} = f(x_t, u_t) = x_t + hu_{1t}\begin{bmatrix} \cos x_{3t} \\ \sin x_{3t} \\ \frac{1}{L}\tan u_{2t} \end{bmatrix}$. We can consider the following nonlinear optimal control problem

$$
\begin{aligned}
\text{minimize } & \sum_{k=1}^{N} \|u_k\|^2 + \gamma \sum_{k=1}^{N-1} \|u_{k+1} - u_k\|^2 \\
\text{subject to } & u_{init} = u_1 \\
& x_{init} = x_1 \\
& x_{t+1} = f(x_t, u_t), \quad t = 2, \ldots, N \\
& x_{final} = f(x_N, u_N),
\end{aligned}
$$

with variables $u_1, \ldots, u_N$ and $x_1, \ldots, x_N$. We use the values $L = 0.1$, $N = 50$, $h = 0.1$ and $\gamma = 10$ and different values of x_{final}. Using augmented Lagrangian algorithm we compute the trajectory and control variables by starting from the same starting point 0 in each case, while the starting point of the input variable u_k is randomly chosen and provided as input. The residual $\left[\frac{f}{g}\right]$ and Jacobians $\left[\frac{Df}{Dg}\right]$ matrices involved with the objective and constraints equations are

$$\underbrace{\left[\begin{array}{c} s_1 \\ \vdots \\ s_N \\ \sqrt{\gamma}(s_2 - s_1) \\ \vdots \\ \sqrt{\gamma}(s_N - s_{N-1}) \\ \phi_1 \\ \vdots \\ \phi_N \\ \sqrt{\gamma}(\phi_2 - \phi_1) \\ \vdots \\ \sqrt{\gamma}(\phi_N - \phi_{N-1}) \\ \hline s_i - s_1 \\ \phi_i - \phi_1 \\ p_{1i} - p_{1,1} \\ p_{1,2} - p_{1,1} - hs_1 \cos\theta_1 \\ \vdots \\ p_{1N} - p_{1,N-1} - hs_{N-1} \cos\theta_{N-1} \\ p_{1f} - p_{1,N} \\ p_{2i} - p_{2,1} \\ p_{2,2} - p_{2,1} - hs_1 \sin\theta_1 \\ \vdots \\ p_{2N} - p_{2,N-1} - hs_{N-1} \sin\theta_{N-1} \\ p_{2f} - p_{2,N} \\ \theta_i - \theta_1 \\ \theta_2 - \theta_1 - h\frac{s_1}{L} \tan\phi_1 \\ \vdots \\ \theta_N - \theta_{N-1} - h\frac{s_{N-1}}{L} \tan\phi_{N-1} \\ \theta_f - \theta_N \end{array}\right]}_{(7N+3)\times 1}, \underbrace{\left[\begin{array}{ccccc} \boldsymbol{I}_N & \mathbf{0}_N & \mathbf{0}_N & \mathbf{0}_N & \mathbf{0}_N \\ \sqrt{\gamma}T_{N-1} & \mathbf{0}_{N-1} & \mathbf{0}_{N-1} & \mathbf{0}_{N-1} & \mathbf{0}_{N-1} \\ \mathbf{0}_N & \boldsymbol{I}_N & \mathbf{0}_N & \mathbf{0}_N & \mathbf{0}_N \\ \mathbf{0}_{N-1} & \sqrt{\gamma}T_{N-1} & \mathbf{0}_{N-1} & \mathbf{0}_{N-1} & \mathbf{0}_{N-1} \\ \hline \begin{bmatrix} -1 & 0 & \\ 0 & 0 & \ldots \end{bmatrix} & \begin{bmatrix} 0 & 0 & \\ -1 & 0 & \ldots \end{bmatrix} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ -h \begin{bmatrix} 0 & \\ \boldsymbol{I}_{\cos\boldsymbol{\theta}} & \\ & 0 \end{bmatrix} & \mathbf{0}_{N+1} & \boldsymbol{T}_{N+1} & \mathbf{0}_{N+1} & h \begin{bmatrix} 0 & \\ \boldsymbol{I}_{\sin\theta} & \\ & 0 \end{bmatrix} \boldsymbol{s} \\ -h \begin{bmatrix} 0 & \\ \boldsymbol{I}_{\sin\boldsymbol{\theta}} & \\ & 0 \end{bmatrix} & \mathbf{0}_{N+1} & \mathbf{0}_{N+1} & \boldsymbol{T}_{N+1} & -h \begin{bmatrix} 0 & \\ \boldsymbol{I}_{\cos\theta} & \\ & 0 \end{bmatrix} \boldsymbol{s} \\ -\frac{h}{L} \begin{bmatrix} 0 & \\ \boldsymbol{I}_{\tan\boldsymbol{\phi}} & \\ & 0 \end{bmatrix} & -\frac{h}{L} \begin{bmatrix} 0 & \\ I_{\sec^2\boldsymbol{\phi}} & \\ & 0 \end{bmatrix} \boldsymbol{s} & \mathbf{0}_{N+1} & \mathbf{0}_{N+1} & \boldsymbol{T}_{N+1} \end{array}\right]}_{(7N+3)\times(5N)},$$

where $T_{N-1} = \underbrace{\begin{bmatrix} -1 & 1 & & & & \\ & -1 & 1 & & & \\ & & -1 & 1 & & \\ & & & \ddots & \ddots & \\ & & & & -1 & 1 \end{bmatrix}}_{(N-1)\times N}$, $T_{N+1} = \underbrace{\begin{bmatrix} -1 & 0 & & & & \\ -1 & 1 & & & & \\ & -1 & 1 & & & \\ & & -1 & 1 & & \\ & & & \ddots & \ddots & \\ & & & & -1 & 1 \\ & & & & & -1 \end{bmatrix}}_{(N+1)\times N}$ and $\boldsymbol{I_y}$ is a diagonal matrix with the values of the vector $\boldsymbol{y}[: N-1]$ on the diagonal.

```
class CarControl:
    def __init__(self, x0, u0, xf):
        self.xi = x0
        self.ui = u0
        self.xf = xf
        self.gm = 10
        self.h = 0.1
        self.L = 0.1
        self.N = 50
        # x = 5XN  (s, ph, p1, p2, th) X N

    def f(self, x):
        s  = x[0*self.N:1*self.N]
        ph = x[1*self.N:2*self.N]
        return np.hstack([
            s,
            np.sqrt(self.gm) * (s[1:]-s[:-1]),
            ph,
            np.sqrt(self.gm) * (ph[1:]-ph[:-1]),
        ])
    def g(self, x):
        s  = x[0 * self.N:1 * self.N]
        ph = x[1 * self.N:2 * self.N]
        p1 = x[2 * self.N:3 * self.N]
        p2 = x[3 * self.N:4 * self.N]
        th = x[4 * self.N:5 * self.N]
        return np.hstack([
            self.ui[0] - s[0],
            self.ui[1] - ph[0],
            self.xi[0] - p1[0],
            (p1[1:]-p1[:-1])-self.h * s[:-1] * np.cos(th[:-1]),
            self.xf[0] - p1[-1],
            self.xi[1] - p2[0],
            (p2[1:] - p2[:-1]) - self.h * s[:-1] * np.sin(th[:-1]),
            self.xf[1] - p2[-1],
            self.xi[2] - th[0],
            (th[1:]-th[:-1]) - self.h/self.L * s[:-1] * np.tan(ph[:-1]),
            self.xf[2]-th[-1],
        ])

    def Df(self, x):
        IN = np.eye(self.N)
        TN = -np.eye(self.N) + np.hstack([np.zeros((self.N, 1)), np.eye(self.N)])[:, :-1]
        TN1=TN[:-1,:]
        sg = np.sqrt(self.gm)
        ZN = np.zeros((self.N, self.N))
        ZN1= ZN[:-1,:]
        return np.vstack([
            np.hstack([IN,      ZN,     ZN,     ZN,     ZN]),
            np.hstack([sg*TN1,  ZN1,    ZN1,    ZN1,    ZN1]),
            np.hstack([ZN,      IN,     ZN,     ZN,     ZN]),
            np.hstack([ZN1,     sg*TN1, ZN1,    ZN1,    ZN1]),
        ])

    def Dg(self, x):
        s  = x[0 * self.N:1 * self.N]
        ph = x[1 * self.N:2 * self.N]
        th = x[4 * self.N:5 * self.N]
        ON1 = np.zeros((2, self.N))
        ON1[0, 0] = -1
        ON2 = np.zeros((2, self.N))
        ON2[1, 0] = -1
        ZR = np.zeros((2, self.N))
        TN = -np.eye(self.N) + np.hstack([np.zeros((self.N, 1)), np.eye(self.N)])[:, :-1]
        TN = np.vstack([np.hstack([-1, np.zeros(self.N-1)]), TN])
        ZN = np.zeros((self.N+1, self.N))
        h, L = self.h, self.L
        def d(a):
            a = np.diag(a)
            a[-1,:] = 0
            return np.vstack([np.zeros(a.shape[0]),a])
        return np.vstack([
            np.hstack([ON1, ON2, ZR, ZR, ZR]),
            np.hstack([-h*d(np.cos(th)),    ZN,                            TN, ZN,  h*d(np.sin(th))*s]),
            np.hstack([-h*d(np.sin(th)),    ZN,                            ZN, TN, -h*d(np.cos(th))*s]),
            np.hstack([-h/L*d(np.tan(ph)), -h/L*d(1/np.cos(ph)**2)*s,   ZN, ZN, TN]),
        ])

    def augmented_lagrangian_algo(self, mu=1, iter_max=100, tol=1e-4):
        x = np.random.randn(5*self.N)
        z = np.zeros(3*self.N+5)
        _f = lambda _x: np.hstack([self.f(_x), np.sqrt(mu) * self.g(_x) + z / 2 / np.sqrt(mu)])
        _Df = lambda _x: np.vstack([self.Df(_x), np.sqrt(mu) * self.Dg(_x)])
        _o = lambda _x: np.linalg.norm(2 * self.Df(_x).T @ self.f(_x) + self.Dg(_x).T @ z)
        res_f = [np.linalg.norm(self.g(x))]
        res_o = [_o(x)]
        res_i = [0]
        mu_i = [mu]
        while iter_max > 0:
            _lm_res = lm(_f, _Df, x, count_max=100)
            x = _lm_res[2][-1]
            z = z + 2 * mu * self.g(x)
            res_f.append(np.linalg.norm(self.g(x)))
            res_o.append(_o(x))
            res_i.append(len(_lm_res[0]))
            mu_i.append(mu)
            if res_f[-1] < tol: break
            if res_f[-1] >= 0.25 * res_f[-2]:
                mu = 2 * mu
            iter_max -= 1
        return x, res_f, res_o, res_i, mu_i, z

# angle vs speed
def plot_as(ax, i):
    x = pd.DataFrame(res[i][0].reshape(5, 50).T, index=range(1, 51),
                     columns=['speed', 'angle', 'p1', 'p2', 'theta'])
    x.iloc[:, :2].plot(ax=ax, grid=True, title=str(xf[i]))
fig = plt.figure()
for i in range(4):
    plot_as(fig.add_subplot(2,2,i+1), i)
plt.tight_layout()

# iteration curve
def plot_ic(ax, i):
    pd.DataFrame({'Feasibility': res[i][1], 'Optimality': res[i][2]}, index=np.cumsum(res[i][3])). \
        plot(ax=ax, grid=True, style='o-', drawstyle="steps-post", logy=True)
fig = plt.figure()
for i in range(4):
    plot_ic(fig.add_subplot(2,2,i+1), i)
plt.tight_layout()
```

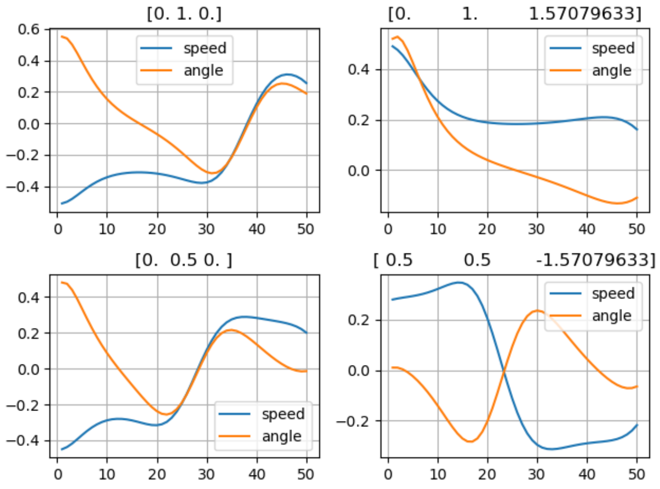

Figure 33: The two inputs: speed and steering angle for the trajectories.

□

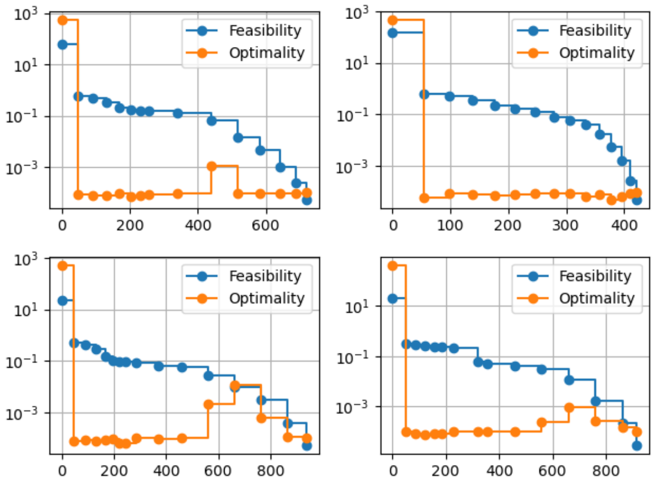

Figure 34: Feasibility and optimality condition residuals in the augmented Lagrangian algorithm for computing the trajectories.

8 Appendix: Some Matrices

8.1 Networks

An $n \times m$ incident matrix for a given directed graph with n vertices and m directed edges is given by $A_{ij} = \begin{cases} -1 & j \to i \\ +1 & i \to j \\ 0 & otherwise \end{cases}$. To represent a network using a graph, if x is an m-vector represent a flow in the network, we interpret x_j as the flow rate along the edge j, with a positive value meaning the flow is in the direction of edge j, and negative meaning the flow is in the opposite direction of edge j. The n-vector $y = Ax$ can be interpreted as the vector of net flows, from the edges, in the nodes: y_i is the total of the flows that come into node i, minus the total of the flows that go out from node i, y_i called the flow surplus at node i. If $Ax = 0$ we say that flow conservation occurs. In this case the flow vector x is called circulation. Sources and sinks can be incorporated by a n-vector s with $s_i > 0$ indicating a source and $s_i < 0$ indicating a sink. The equation $Ax + s = 0$ represents the conservation in this case.

For v being an n-vector for potential at each node in the graph we can write $\underset{m\times 1}{u} = \underset{(m\times n)}{A^T} \underset{n\times 1}{v}$, where u gives the potential difference across the edges $u_j = v_l - v_k$ where edge goes from node k to node l. When this potential difference is small across the edges we can represent it as $\mathcal{D}(v) = \|A^T v\|^2$ the *Dirichlet energy* or Laplacian quadratic form associated with the graph. $\mathcal{D}(v) = \sum_{edges\ (k,l)} (v_l - v_k)^2$, which is the sum of the squares of the potential differences of v across all edges in the graph. A set of node potentials with small Dirichlet energy can be thought of as smoothly varying across the graph. It arises as a measure of roughness in a network. For a chain graph, or a time series, Dirichlet energy is related to the variance of the series.

8.2 Convolution

The convolution of an n-vector a and an m-vector b is the $(n + m - 1)$-vector denoted by $c = a * b$, with entries $c_k = \sum_{i+j=k+1} a_k b_j$, $k = 1, \ldots, n + m - 1$. If a and b represent the coefficients of two polynomials $p(x) = a_1 + a_2 x + \ldots + a_n x^{n-1}$, $q(x) = b_1 + b_2 x + \ldots + b_m x^{m-1}$, then the coefficients of the product polynomial $p(x)q(x)$ are represented by $c = a * b$, i.e. c_k is the coefficient of x^{k-1} in $p(x)q(x)$. Convolution is symmetric and associative. For a fixed a or b $a * b$ is a linear functions. This means we can write $a * b = T(b)a = T(a)b$, where $T(b)$ is the $(n+m-1) \times n$ matrix with entries $T(b)_{ij} = \begin{cases} b_{i-j+1} & 1 \le i - j + 1 \le m \\ 0 & otherwise \end{cases}$ and similarly for $T(a)$. For

example, with $n = 4$ and $m = 3$, we can see it in a matrix form $T(b) = \begin{bmatrix} b_1 & 0 & 0 & 0 \\ b_2 & b_1 & 0 & 0 \\ b_3 & b_2 & b_1 & 0 \\ 0 & b_3 & b_2 & b_1 \\ 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & b_3 \end{bmatrix}$, $T(a) = \begin{bmatrix} a_1 & 0 & 0 \\ a_2 & a_1 & 0 \\ a_3 & a_2 & a_1 \\ a_4 & a_3 & a_2 \\ 0 & a_4 & a_3 \\ 0 & 0 & a_4 \end{bmatrix}$.
The matrices $T(b)$ and $T(a)$ are called *Toeplitz* matrices, which means the entries on any diagonal ($i-j$ constant) are the same. Application includes time series averaging, first order differencing. *Fast convolution algorithm* based on fast Fourier transform has a runtime of $\mathcal{O}((m+n)\log(m+n))$ with no additional memory requirement beyond $m+n$ numbers.

We can extend it to multiple dimensions. Suppose that A is an $m \times n$ matrix and B is an $p \times q$ matrix. Their convolution is the $(m+p-1) \times (n+q-1)$ matrix $C_{rs} = \sum_{i+k=r+1, j+l=s+1} A_{ij}B_{kl}$, $r = 1, \dots, m+p-1$, $s = 1, \dots, n+q-1$, where the indices are restricted to their ranges, this is denoted by $A \star B$.

Example 18. We take an images represented by $m \times n$ matrix X and use $Y = X \star B$ to get the effect of blurring the image by the point spread function (PSF) given by the entries of the matrix B. With $B = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}$, $Y = X \star B$ is an image where each pixel value is the average of a 2×2 block of 4 adjacent pixels in X. Image Y would be perceived as the image X, with some blurring of the fine details. With the point spread function $D^{hor} = \begin{bmatrix} 1 & -1 \end{bmatrix}$ the pixel values in the image $Y = X \star D^{hor}$ are the horizontal first order differences of those in X, $Y_{ij} = X_{ij} - X_{i,j-1}$, $i = 1, \dots, m$, $j = 2, \dots, n$. With the point spread function $D^{ver} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ the pixel values in the images $Y = X \star D^{ver}$ are the vertical first order differences of those in X: $Y_{ij} = X_{ij} - X_{i-1,j}$, $i = 2, \dots, m$, $j = 1, \dots, n$.

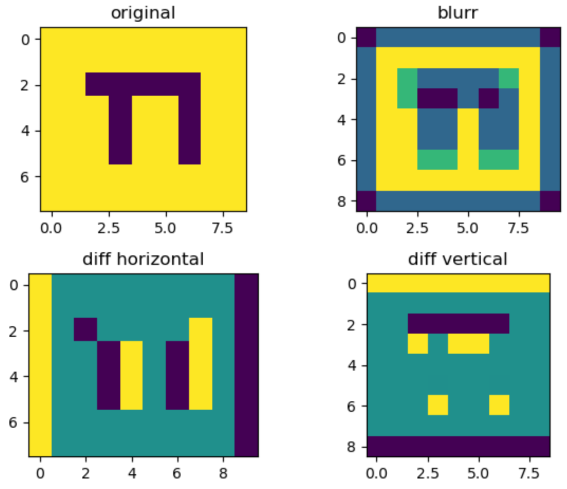

Figure 35: The original image and the convolutions.

```
from scipy.signal import fftconvolve

X = \
"""111111111
111111111
110000011
111011011
111011011
111011011
111111111
111111111"""
X = np.array([[int(j) for j in i.strip()] for i in X.split('\n')])
B = np.array([[0.25, 0.25],
              [0.25, 0.25]])
Dhor = np.array([[1,-1]])
```

```
Dver = np.array([[1],[-1]])
fig = plt.figure()

ax1 = fig.add_subplot(221)
plt.imshow(X)
ax1.set_title('original')
ax2 = fig.add_subplot(222)
plt.imshow(fftconvolve(X,B))
ax2.set_title('blurr')
ax3 = fig.add_subplot(223)
plt.imshow(fftconvolve(X,Dhor))
ax3.set_title('diff horizontal')
ax4 = fig.add_subplot(224)
plt.imshow(fftconvolve(X,Dver))
ax4.set_title('diff vertical')
plt.tight_layout()
```

□

8.3 Matrix operations

Matrix multiplication of compatible matrices is, in general, not commutative, i.e $AB \neq BA$. Also, $\langle y, Ax\rangle = \langle A^T y, x\rangle$. For any matrix A, the matrix $A^T A$ is called the Gram matrix which is symmetric. The complexity of matrix multiplication is $\mathcal{O}(mnp)$.