TP 5 – AOS1
Regularization
Corrigé

In the following, we will study the differences between the classical linear regression model and their most popular regularized versions: the ridge regression model and the Lasso model.

We will use the sklearn Python library and some tools from the scipy library.

```
from sklearn.linear_model import Ridge, LinearRegression
from scipy.linalg import svd
```

A custom class is provided in the file utils.py that models a linear random phenomena. You have to first create the random phenomena

```
from utils import Event
evt = Event(n_features=n_features, effective_rank=3, noise_level=1)
```

Then you can repeatedly sample from it

```
batch1_X, batch1_y = evt.sample(n_samples=100)
batch2_X, batch2_y = evt.sample(n_samples=200)
```

More formally the following model is implemented inside Event

$$y = X\beta + \varepsilon.$$

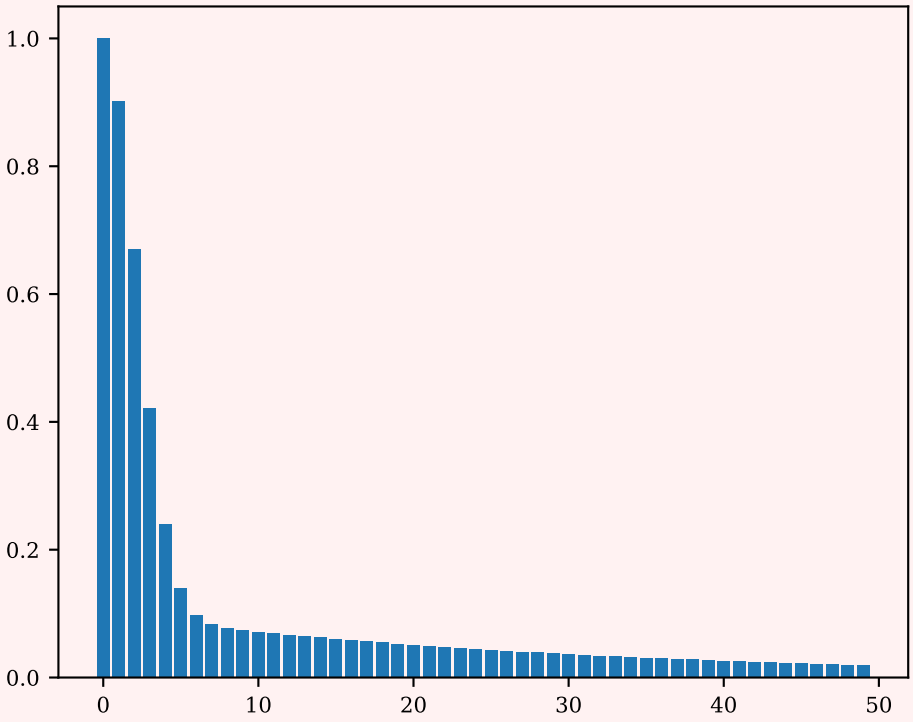
The coefficients are randomly generated unless given as argument to `Event`. They are accessible with `evt.coefficients`.

The matrix X is created on demand when getting samples with the `sample` method. It is controlled by the argument `n_features` (10 features by default) and the `effective_rank` integer that is the "denoised rank" of X.

The standard deviation σ of the noise ε is controlled by the noise_level parameter in Event. More precisely, noise_level indicates a ratio in percentage between σ and $\|\boldsymbol{\beta}\|$. The standard deviation is available in the attribute model_noise.

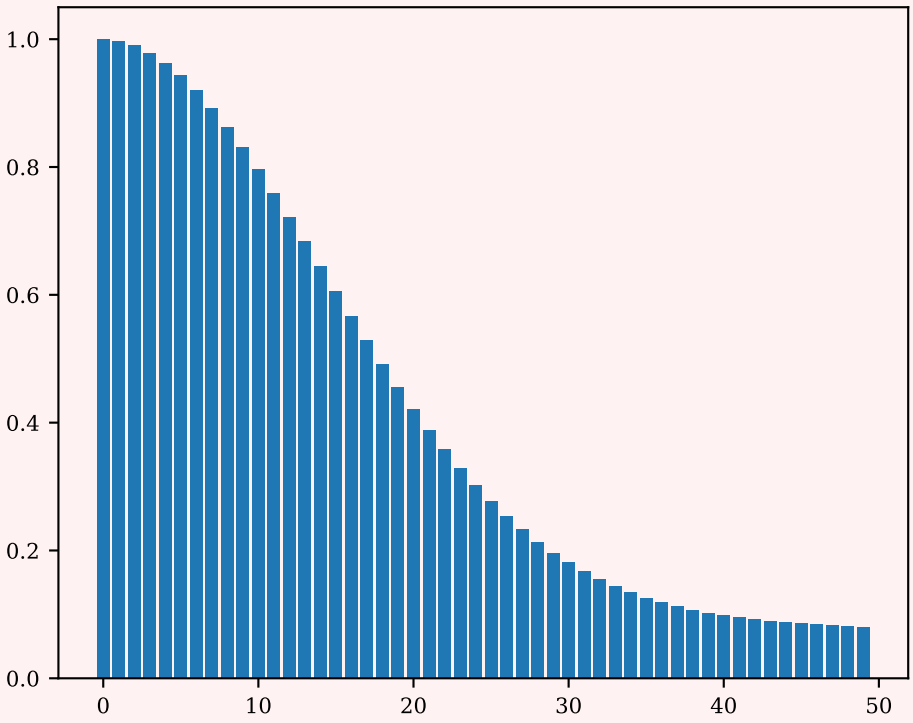
(1) Using a SVD, compute the singular values d_i, $1 \leqslant i \leqslant k$ of X. What is the effect of the parameter `effective_rank`? What is the effective rank supposed to model ?

In [1]:
```
import scipy.linalg as linalg

# Load Event class, you can copy-paste!
import sys
sys.path.append('.')
from src.utils import Event

n_features = 50
evt = Event(n_features=n_features, effective_rank=3)
X, y = evt.sample()
U, s, VT = linalg.svd(X, full_matrices=False)

plt.bar(range(len(s)), s)
plt.show()
```



The rate of decay of singular values is high. There seem to be 3 or 4 significant singular values. After that they are exponentially decreasing.

In [2]:
```
evt = Event(n_features=n_features, effective_rank=20)
X, y = evt.sample()
U, s, VT = linalg.svd(X, full_matrices=False)
plt.bar(range(len(s)), s)
plt.show()
```



Second plot shows a decaying rate of singular values that is much smaller and it shows a change in curvature around 20, the chosen number of effective rank.
The effective rank is then the underlying number of factors that explains the event we sample from.

(2) By repeatedly fitting each model on the same Event object, give an estimate of the bias, variance and risk of the estimator at a chosen point.

In [3]:
```
import numpy as np
from sklearn.linear_model import Ridge, LinearRegression, Lasso,
↪  LassoCV, RidgeCV
n_features = 50
n_samples = 200
evt = Event(n_features=n_features, effective_rank=3,
↪  noise_level=1)
X0 = np.random.randn(1, n_features)
ntimes = 10
```

Define a function `risk` that is returning an estimation of the variance, bias and risk of prediction at point `X0`.

In [4]:
```
from sklearn.utils._testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning

def compute_risk(model, evt, ntimes, n_samples, X0):
    pred_model = []
    y0 = np.dot(X0, evt.coefficients)
    for i in range(ntimes):
        X, y = evt.sample(n_samples=n_samples)
        with ignore_warnings(category=ConvergenceWarning):
            model.fit(X, y)
        pred = model.predict(X0)
        pred_model.append(pred)
    bias = np.mean(pred_model) - y0
    var = np.var(pred_model)
    y0s = y0 + evt.model_noise * np.random.randn(ntimes)
    risk = np.mean((pred_model - y0s)**2)
    return (bias, var, risk)

risk_lin = compute_risk(LinearRegression(), evt, ntimes,
↪  n_samples, X0)
risk_ridge = compute_risk(Ridge(alpha=0.1), evt, ntimes,
↪  n_samples, X0)
risk_lasso = compute_risk(Lasso(alpha=0.1), evt, ntimes,
↪  n_samples, X0)
output = "Bias: %f\tVariance: %f\tRisk: %f"
print("Linear regression: ", output % risk_lin)
```

Out [4]:
```
Linear regression:  Bias: 27.264046        Variance:
↪  10263.238850        Risk: 10999.102687

<string>:1: DeprecationWarning: Conversion of an array with ndim
↪  > 0 to a scalar is deprecated, and will error in future.
↪  Ensure you extract a single element from your array before
↪  performing this operation. (Deprecated NumPy 1.25.)
```

In [5]:
```
print("Ridge regression: ", output % risk_ridge)
```

Out [5]:
```
Ridge regression:  Bias: 26.170951        Variance:
↪  5.012061        Risk: 679.652267
```

In [6]:
```
print("Lasso regression: ", output % risk_lasso)
```

Out [6]:
```
Lasso regression:  Bias: 28.532400        Variance:
↪  0.008043        Risk: 794.328577
```

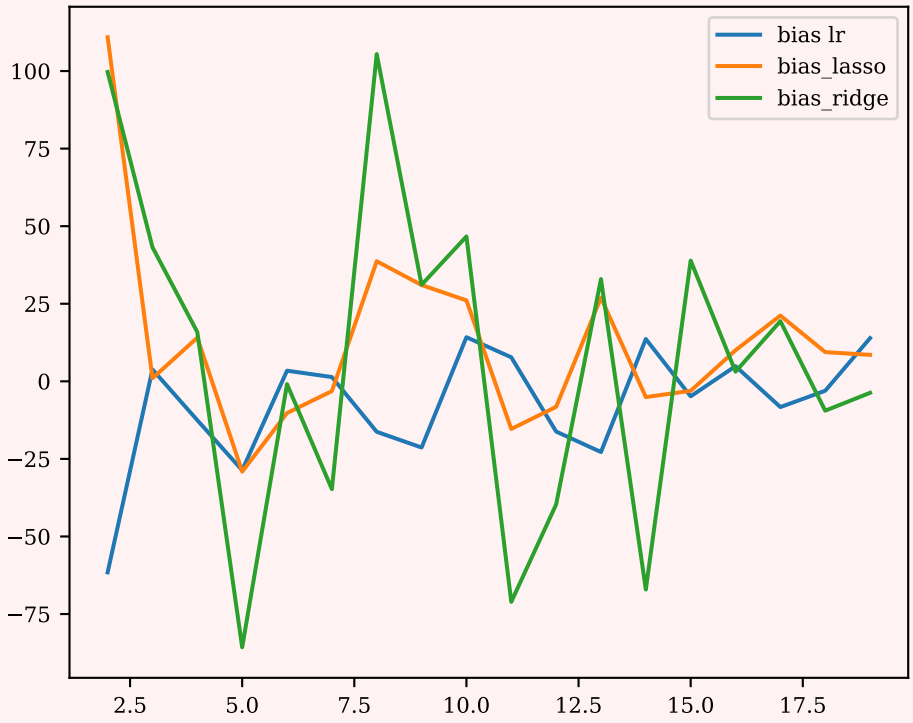
(3) Show the influence of the dataset's effective rank on the 3 algorithms and interpret.

In [7]:
```
X, y = evt.sample(n_samples=100)
with ignore_warnings(category=ConvergenceWarning):
    lasso_alpha = LassoCV(cv=5, random_state=0).fit(X, y).alpha_
    print('Best alpha for lasso is %f' % lasso_alpha)
    ridge_alpha = RidgeCV(alphas=np.logspace(1e-3, 10),
    ↪  cv=5).fit(X, y).alpha_
    print('Best alpha for ridge is %f' % ridge_alpha)
```

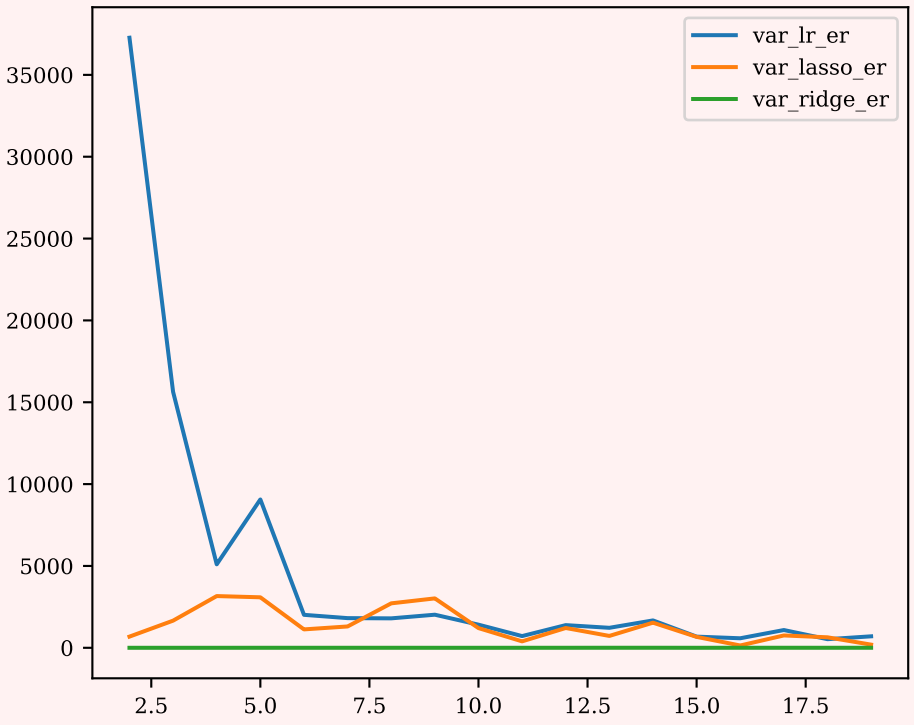
Out [7]:
```
Best alpha for lasso is 0.000125
Best alpha for ridge is 1.002305
```

In [8]:
```
effective_ranks = range(2, 20)

bias_lr_er = []
var_lr_er = []
risk_lr_er = []
bias_ridge_er = []
var_ridge_er = []
risk_ridge_er = []
bias_lasso_er = []
var_lasso_er = []
risk_lasso_er = []

for effective_rank in effective_ranks:
    evt = Event(n_features=n_features,
    ↪  effective_rank=effective_rank, noise_level=1)
    bias_lr, var_lr, risk_lr = compute_risk(LinearRegression(),
    ↪  evt, ntimes, n_samples, X0)
    bias_lr_er.append(bias_lr)
    var_lr_er.append(var_lr)
    risk_lr_er.append(risk_lr)
    bias_ridge, var_ridge, risk_ridge =
    ↪  compute_risk(Ridge(alpha=ridge_alpha), evt, ntimes,
    ↪  n_samples, X0)
    bias_ridge_er.append(bias_ridge)
    var_ridge_er.append(var_ridge)
    risk_ridge_er.append(risk_ridge)
    bias_lasso, var_lasso, risk_lasso =
    ↪  compute_risk(Lasso(alpha=lasso_alpha), evt, ntimes,
    ↪  n_samples, X0)
    bias_lasso_er.append(bias_lasso)
    var_lasso_er.append(var_lasso)
    risk_lasso_er.append(risk_lasso)
```

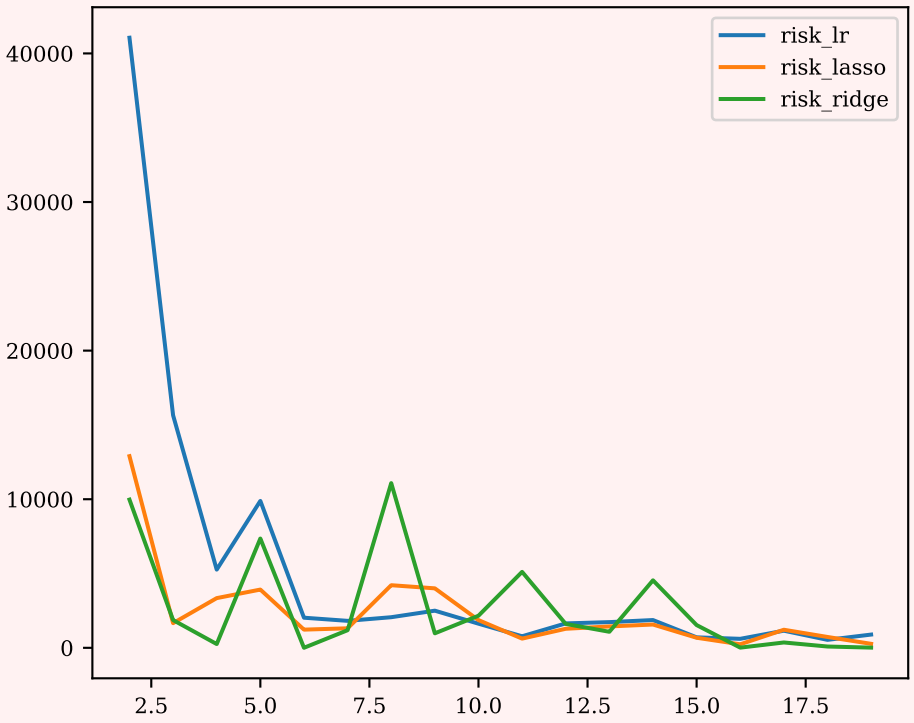
In [9]:
```
plt.plot(effective_ranks, bias_lr_er, label='bias lr')
plt.plot(effective_ranks, bias_lasso_er, label='bias_lasso')
plt.plot(effective_ranks, bias_ridge_er, label='bias_ridge')
plt.legend()
plt.show()
```



In [10]:
```
plt.figure()
plt.plot(effective_ranks, var_lr_er, label='var_lr_er')
plt.plot(effective_ranks, var_lasso_er, label='var_lasso_er')
plt.plot(effective_ranks, var_ridge_er, label='var_ridge_er')
plt.legend()
plt.show()
```



In [11]:
```
plt.figure()
plt.plot(effective_ranks, risk_lr_er, label='risk_lr')
plt.plot(effective_ranks, risk_lasso_er, label='risk_lasso')
plt.plot(effective_ranks, risk_ridge_er, label='risk_ridge')
plt.legend()
plt.show()
```



As expected, the linear regression gives an unbiased estimator that has a high variance when the effective rank is small.
On the contrary, the ridge regression and lasso always have a nonzero bias but a smaller variance when the effective rank is small. The risk combining the bias and the variance is smaller for ridge regression or lasso.

Problem: make elastic net outshine the lasso

The elastic net regression was invented to compensate for the lack of robustness of the lasso regression. The elastic net especially outshines the lasso when some variables are highly correlated and on the same scale. You have been shown in the last course's slides some intuitive arguments demonstrating why it should particularly be more robust in this case. But could you demonstrate this experimentally?

Design a regression dataset in which we want to select variables and where the elastic net gives better results in terms of stability of the set of selected variables.