benign-overfitting

January 9, 2022

BENIGN OVERFITTING

This is inspired by Peter Bartlett's lecture, but it's not a direct demonstration of his claim. His claim is already proven, so I want to instead illustrate the essential idea in an extremely (read: "uselessly") simple setting, just to gain intuition.

The idea is that the Moore-Penrose pseudoinverse gives rise to a linear regression estimator that, among all best-fitting β's, chooses the one with smallest norm. When the number of regressors is smaller than the number of observations, this recovers the usual (unique) OLS estimator. But when it is much larger, the output is something very similar-looking to a ridge-regularized linear regression.

[1]:
```
from typing import Tuple, Optional
import scipy.stats as stats
import pandas as pd
import numpy as np
from numpy.linalg import pinv
import sklearn.linear_model as lm
```

1 Data-generating process

Exceedingly simple. There are `N` raw-data feeds. Each feed is an i.i.d. standard Normal random variable. Of these, `N` feeds, `M` are actually relevant to our `y`-generating process, in the simplest way: `y` is the sum of the `M` relevant data feeds, plus white noise. Exactly which `M` are relevant is random. We have `T` observations in the training sample, and, because it doesn't matter, also `T` in the test sample.

We have $M \ll N$ and also $T \ll N$, but the relevant thing here is that $T \ll N$. Usually, we'd say that blindly using all `N` raw-data feeds is a terrible idea, because we'll be able to perfect interpolate (i.e. perfect overfit!) the training data. BUT… is that conventional wisdom actually true?

[2]:
```
DataSubsample = Tuple[pd.DataFrame, pd.Series]
# train, test
DataSample = Tuple[DataSubsample, DataSubsample]

N = 10_000
M = int(N/10)
T = int(N/100)
NOISE = 1
```

```python
def _get_x_name(base: str="x", n: int=0) -> str:
    return f"{base}_{n}"

def ___gen_x(T: int=T) -> pd.Series:
    """Generate observations for a single raw-data feed."""
    x = stats.norm.rvs(size=T)
    x = pd.Series(x)
    return x

def __gen_X(N: int=N, T: int=T) -> pd.DataFrame:
    """Generate observations for all raw-data feeds."""
    X = {_get_x_name(n): ___gen_x() for n in range(N)}
    X = pd.DataFrame(X)
    assert X.shape == (T, N), X.shape
    return X

def _pick_relevant_x(N: int=N, M: int=M) -> pd.Series:
    """beta[n] indicates whether X[:,n] is relevant."""
    beta_star = pd.Series(0, index=range(N))
    # detereministically pick the first `M` of these, just so we
    # have the correct total number of "live" picks to distribute
    beta_star.iloc[:M] = 1
    # distribute the picks uniformly randomly
    beta_star = beta_star.sample(n=N).reset_index(drop=True)
    beta_star = beta_star.rename(index=_get_x_name)
    assert beta_star.sum() == M, beta_star.sum()
    return beta_star

def __get_y(beta_star: pd.Series, X: pd.DataFrame) -> pd.Series:
    """Generate y's, where beta_star indicates whether a column is relevant."""
    # pick out the relevant x's, zero out the rest
    y = X * beta_star
    # y is a simple sum of the relevant raw-data feeds
    y = y.sum(axis="columns")
    # add noise
    y = y + stats.norm.rvs(scale=NOISE, size=T)
    y.name = "y"
    return y


def _gen_data_subsample(beta_star: pd.Series) -> DataSubsample:
    """Get X, y."""
    X = __gen_X()
    y = __get_y(beta_star=beta_star, X=X)
    return X, y
```

```
def gen_data_sample() -> Tuple[pd.Series, DataSample]:
    beta_star = _pick_relevant_x()
    # train subsample
    X, y = _gen_data_subsample(beta_star=beta_star)
    train_subsample = X, y
    # test subsample
    X_, y_ = _gen_data_subsample(beta_star=beta_star)
    test_subsample = X_, y_
    sample = train_subsample, test_subsample
    return beta_star, sample

np.random.seed(1337)
# M relevant x's, training subsample, testing subsample
beta_star, ((X, y), (X_, y_)) = gen_data_sample()
```

2 Feature creation

To show how performance varies with increasing degree of overfitting, we need to increase the degree of overfitting. To do this in the linear regression setting, we can just increase the number of regressors until it equals—and indeed even further until it is much larger than—the number of observations. We'll do this in a very simple way: If we need `R` regressors, we'll use the first `R` columns of `X`. Notice that this doesn't matter: The `N` raw-data feeds were i.i.d. random, and which `M` of the `N` raw-data feeds were chosen to be relevant was also uniformly random. So shuffling the `N` raw-data feeds before choosing `R` of them is unnecessary.

[3]:
```
"""
# this implementation uses
# (a flavor of) random Fourier features,
# which would be much more useful for actual prediction.
# however, it's already getting to complicated
# for my liking,
# and IMO obscures the magic.

def _get_rhs(X: pd.DataFrame) -> pd.Series:
    w = stats.norm.rvs(size=len(X.columns))
    w = pd.Series(w, index=X.columns)
    rhs = X * w
    rhs = rhs.sum(axis="columns")
    rhs = np.sin(rhs)
    return rhs

def get_RHS(X: pd.DataFrame, R: int) -> pd.DataFrame:
    RHS = {_get_x_name(base="r", n=r): _get_rhs(X=X) for r in range(R)}
    RHS = pd.DataFrame(RHS)
    return RHS
"""
```

```
def get_RHS(X: pd.DataFrame, R: int) -> pd.DataFrame:
    return X.iloc[:, :R]
```

3 Model estimation

[4]:
```
RIDGE_REGULARIZATION = 1


def __fit_star_(**kwargs) -> pd.Series:
    """Ignored args and returns beta_star."""
    return beta_star

def __fit_bo(y: pd.Series, RHS: pd.DataFrame) -> pd.Series:
    """Benignly-overfit model:
    (X'X)^{-1} X'y, where ^{-1} denotes Moore-Penrose pseudoinverse.
    This is OLS if len(X.columns) <= len(y).
    """
    X_T_X = RHS.T @ RHS
    X_T_X_inv = pinv(X_T_X)
    X_T_y = RHS.T @ y
    beta_hat = X_T_X_inv @ X_T_y
    beta_hat = pd.Series(beta_hat, index=RHS.columns)
    return beta_hat

def __fit_ridge(
        y: pd.Series, RHS: pd.DataFrame,
        regularization: float=RIDGE_REGULARIZATION
    ) -> pd.Series:
    """This is OLS if penalty == 0."""
    model = lm.Ridge(alpha=regularization, fit_intercept=False)
    model = model.fit(y=y, X=RHS)
    beta_hat = pd.Series(model.coef_, index=RHS.columns)
    return beta_hat


def _fit_model(y: pd.Series, RHS: pd.DataFrame, kind: str="star") -> pd.Series:
    """Get beta_hat."""
    _fit = {
        "star": __fit_star_,
        "bo": __fit_bo,
        "ridge": __fit_ridge
    }[kind]
    beta_hat = _fit(y=y, RHS=RHS)
    return beta_hat
```

```
def fit_model(
        y: pd.Series, X: pd.DataFrame, R: int, kind: str="star"
    ) -> pd.Series:
    RHS = get_RHS(X=X, R=R)
    beta_hat = _fit_model(y=y, RHS=RHS, kind=kind)
    # fill the potentially-empty var's
    beta_hat = beta_hat.reindex(index=X.columns).fillna(0)
    return beta_hat

def fit_model_(R: int, kind: str="star"):
    return fit_model(y=y, X=X, R=R, kind=kind)
```

4 Model evaluation

[5]:
```
def __get_y_hat(beta: pd.Series, X: pd.DataFrame) -> pd.Series:
    y_hat = X * beta
    y_hat = y_hat.sum(axis="columns")
    return y_hat

def __get_loss(y: pd.Series, y_hat: pd.Series) -> float:
    """RMSE."""
    loss = y_hat - y
    loss = loss**2
    loss = loss.mean()
    loss = loss**0.5
    return loss

def _get_loss(y: pd.Series, X: pd.DataFrame, beta: pd.Series) -> float:
    y_hat = __get_y_hat(beta=beta, X=X)
    return __get_loss(y=y, y_hat=y_hat)

def get_loss(
        y: pd.Series, X: pd.DataFrame,  # train subsample
        y_: pd.Series, X_: pd.Series,  # test subsample
        R: int,
        kind: str="star"
    ) -> float:
    """Fit specified model and get test-subsample loss."""
    beta_hat = fit_model(y=y, X=X, R=R, kind=kind)
    loss = _get_loss(y=y_, X=X_, beta=beta_hat)
    return loss

def get_loss_(R: int, kind: str="star") -> float:
    return get_loss(y=y, X=X, y_=y_, X_=X_, R=R, kind=kind)
```

5 Results

In this oversimplified toy setup, the actual test-subsample performance of the models is almost indistinguishable from (indeed worse than) a model that simply ignores the x's and guesses 0 every time (`flat loss`), and nowhere near the performance of the ground-truth model (`star loss`).

Nevertheless, it shows the point: As $R \to T$, that is, as the number of regressors gets very close to the number of observations, we suffer from a singularity issue, in particular when R approaches T from above. The RMSE of the OLS-type estimator blows up (`bo loss`), and we see the canonical benefit of regularization (in this case, a no-thought ridge model) (`ridge loss`). Yet, when $R \gg T$, that is, when the number of regressors is much greater than the number of observations, the OLS-type model actually becomes indistinguishable from the regularized model! So, thanks to the Moore-Penrose pseudoinverse, as we really crank up the complexity (dimensionality) of our model, we get a natural implicit regularization, without ever having to consciously choose any regularization parameters.

Note: To actually get some good performance out of this, I think we could make the methodology closer to Bartlett's, e.g. use random Fourier features to have a better chance of "blending in" some of the predictive features.

[6]:
```
print("R=0:")
print(f"star loss = {get_loss_(R=None)}")
print(f"flat loss = {(y_**2).mean()**0.5}")
print()

for R in (1, 5, int(T/2), T-1, T, T+1, int((T+M)/2), M):
    print(f"R={R}:")
    for kind in ("bo", "ridge"):
        print(f"{kind} loss = {get_loss_(R=R, kind=kind)}")
    print()
```

```
R=0:
star loss = 1.017412008907012
flat loss = 30.54245569842787

R=1:
bo loss = 30.866563472420818
ridge loss = 30.861803801717798

R=5:
bo loss = 32.93418422764194
ridge loss = 32.90684156832083

R=50:
bo loss = 41.344341229732294
ridge loss = 40.79174373136495

R=99:
bo loss = 205.9078525698844
```

```
ridge loss = 80.08616246165958

R=100:
bo loss = 504.68851633075343
ridge loss = 80.47778433265394

R=101:
bo loss = 403.8034618866445
ridge loss = 78.47148590195188

R=550:
bo loss = 34.891655936046845
ridge loss = 34.87125980639601

R=1000:
bo loss = 33.08875089562013
ridge loss = 33.083695233986575
```