Topics in Econometrics and Statistics

Tim Mensinger

Sep 06, 2020

CONTENTS

Topics in Econometrics and Statistics, University of Bonn, 2020 – Prof. Joachim Freyberger

In this document I present my submission to the project for the topics class in econometrics and statistics, due by 13th September 2020.

CHAPTER

ONE

PROBLEM DESCRIPTION

For the project we have been given a simulated data set and a real data set. For each data set some of the observed outcomes are held back. The task is to build a model for each data set on the training set and submit the predictions on the test set.

The following part of this document is structured as follows. At the end of this section I present a link to my predictions for the respective data sets. In the next chapter I present the model for the simulated data set. Aftwards I introduce my model for the stock data set.

CHAPTER

TWO

DATA FORMAT

Please note that I changed the file format from an RData file to a parquet file, as this is not dependent on the specific programming language. If you want to reproduce my results using my code (see below) you need to transform the RData files to parquet files. This can easily be done, for example, with the SparkR package for R. For the project to run I expect the two data files to be located in the folder `data` with names `simulated_data.parquet` and `stock_data.parquet`, respectively.

CHAPTER

THREE

CODE

In this project I have to sets of code bases. Nearly all of the code presented in the following notebooks is embedded in the notebooks themselves. For the final predictions however, I use Python scripts which are stored here. The script build_project.py runs all scripts that are necessary to produce the predictions. If you want to rerun these codes on your computer open your favorite terminal emulator and run the following line by line (I assume you have at least miniconda already installed on your system, otherwise read the note below.)

```
$ git clone https://github.com/timmens/topics-project.git
$ conda env create -f environment.yml
$ conda activate topics-project
$ cd codes
$ python build_project.py
```

Note. It is not necessary to use conda here as long as all the packages that I use are available to the Python interpreter. Conda just provides a very easy way to create a sandbox environment in which all packages will be available without messing with the system.

CHAPTER

FOUR

PREDICTIONS

Predictions for the respective data sets are stored on github and can be downloaded here

- Simulated data: (click here to download)
- Stock data: (click here to download)

4.1 Simulated Data

In the following sections I present my investigation of the simulated data set. I start with a quick exploration of the data set. I then show my first endeavors to analyze the data using reverse engineering. And finally I show my top four models which I compare on a validation set.

4.1.1 Data Description

Let us first look at a few observations of the data set.

```
import os
from pathlib import Path
import pandas as pd
ROOT = Path(os.getcwd()).parent
```

```
df = pd.read_parquet(ROOT / "data" / "simulated_data.parquet")
df.iloc[:5, :10]
```

```
          Y        X1        X2        X3        X4        X5        X6  \
0  2.125298  0.711338  0.683167  0.493706  0.317948  0.374013  0.600761
1 -1.102707  0.666355  0.422247  0.366055  0.390193  0.499254  0.602870
2 -2.847834  0.065469  0.307784  0.225924  0.233217  0.528559  0.468785
3 -0.386088  0.715237  0.487894  0.716503  0.595477  0.619713  0.761047
4  2.518977  0.144925  0.368206  0.335244  0.426445  0.564264  0.478321

         X7        X8        X9
0  0.630743  0.726537  0.632634
1  0.379105  0.547651  0.507543
2  0.787321  0.831501  0.957693
3  0.879549  0.925856  0.979782
4  0.466601  0.426546  0.591990
```

The data set consists of 100_000 observations of a single continuous outcome and 100 continuous features which have been transformed to a uniform distribution. Of the 100_000 observations 20_000 are designed for the testing step and are marked by a `NaN` in the outcome column.

Train / Validation Split

I (randomly) split the remaining 80_000 *labelled* data points into 65_000 (81.25%) training points and 15_000 validation points. As is standard in the literature I will train all of my models on the training points and compare the performance on the validation points. The *best* model overall is then trained on all 80_000 points and used to predict the outcomes on the test set. This splitting procedure is implemented in the script train_test_split.py.

Next Up

In the next section on "reverse engineering" I will present a few techniques I used to learn more about the data at hand. If you only care about the final model I considered then feel free to skip this section.

4.1.2 Reverse Engineering

In the following I will present a few things I learned about the data through "*reverse data engineering*". The main idea was to learn enough of the underlying process to be able to propose simple models from which I could simulate data, which in turn I could compare to the observed data to test the fit. Unfortunately however, in the end I did not learn enough to beat my final black-box algorithms in terms of prediction on the validation set, which is why I did not pursue this path any further. Nevertheless the insights made here can still be interesting. If, however, you wish to jump right to the final model please skip this section.

Preliminaries

```
import os
import importlib
from pathlib import Path

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import lars_path
from sklearn.feature_selection import RFECV
from sklearn.preprocessing import PolynomialFeatures
from sklearn.gaussian_process.kernels import Matern

from scipy.stats import multivariate_normal
from scipy.stats import norm
from scipy.stats import t

from IPython.display import Video

ROOT = Path(os.getcwd()).parent

# load module using path (I am to lazy to make a python package out of this project..)
spec = importlib.util.spec_from_file_location("shared.py", str(ROOT / "shared.py"))
shared = spec.loader.load_module()
```

(continues on next page)

(continued from previous page)

```
plt.rcParams['figure.figsize'] = [10, 6]
colors = ["#DAA520", "#4169E1", "#228B22", "#A52A2A"]
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 2.5})
sns.set_palette(sns.color_palette(colors))
sns.set_style("whitegrid")
```

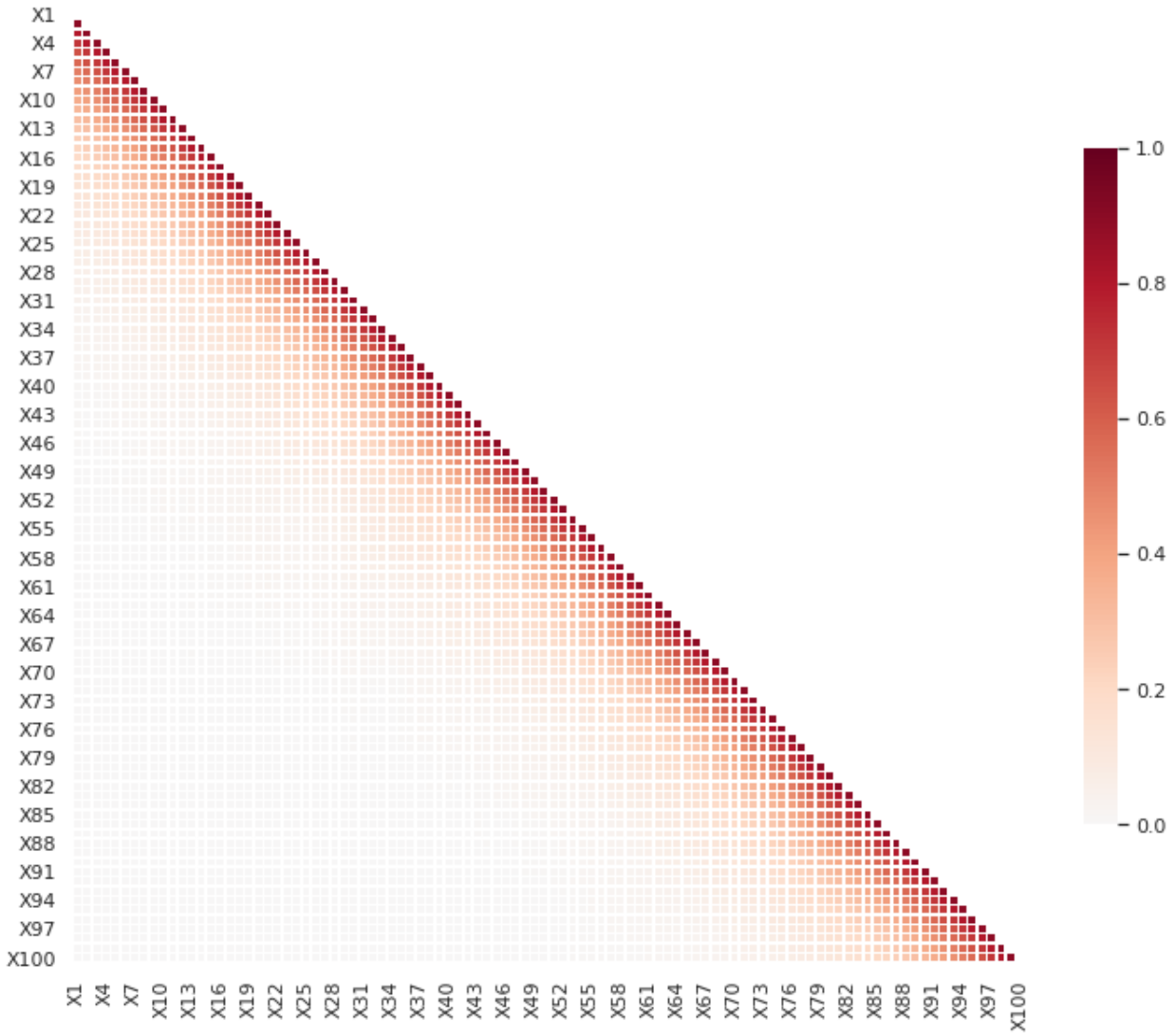
```
df = pd.read_parquet(ROOT / "bld" / "train_simulated.parquet")
X = df.drop("Y", axis=1)
y = df["Y"]
```

Features

By construction our data set consists of $K = 100$ features which are transformed to be uniformly distributed on $[0, 1]$. Unfortunately this is all the prior knowledge we have.

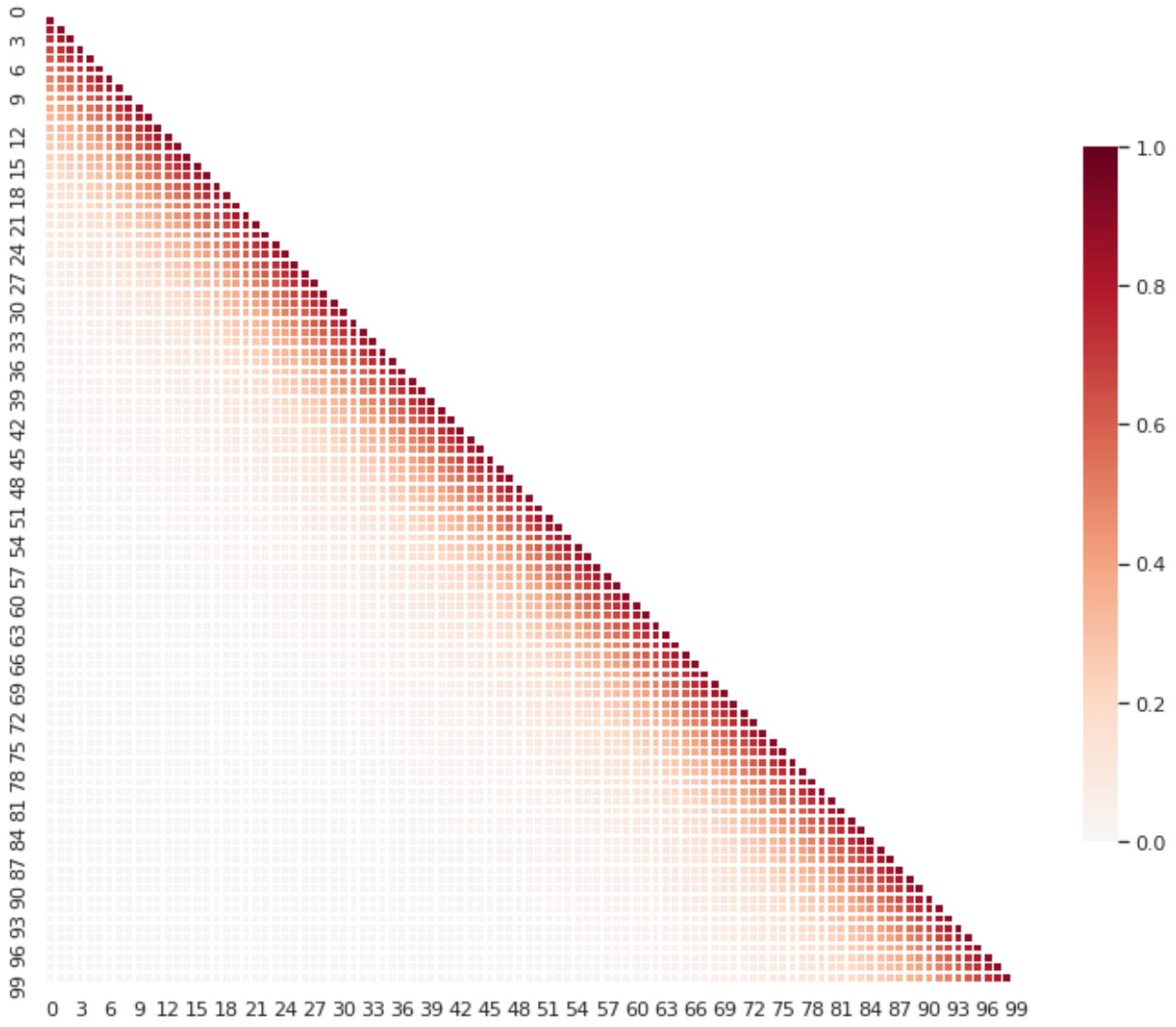
Looking at the correlation between features we notice that the correlation structure looks very similar to the covariance structure of a Matérn covariance function. This can be seen especially well when considering a correlation heatmap of the observed features.

```
shared.correlation_heatmap(df=X)
```



For a comparison, consider a (true) Matérn covariance heatmap.

```
matern_kernel = Matern(length_scale=0.1, nu=0.5)

x = np.linspace(0, 1, 100).reshape(100, -1)
Sigma = matern_kernel(x)

shared.correlation_heatmap(corr=Sigma)
```



Simulation?

Given the hypothesis that the features were simulated using the Matern covariance matrix, can we actually formulate an algroithm for this simulation step?

Let Z denote a K dimensional random vector. If we assume that the individual entries Z_k have unit standard deviation, then $\text{corr}(Z) = \text{cov}(Z)$.

Hence, we can simulate features with the structure from above using the following simple algorithm. For this let Σ denote the above Matérn covariance matrix (parameterized with $\ell = 0.1$ and $\nu = 0.5$).

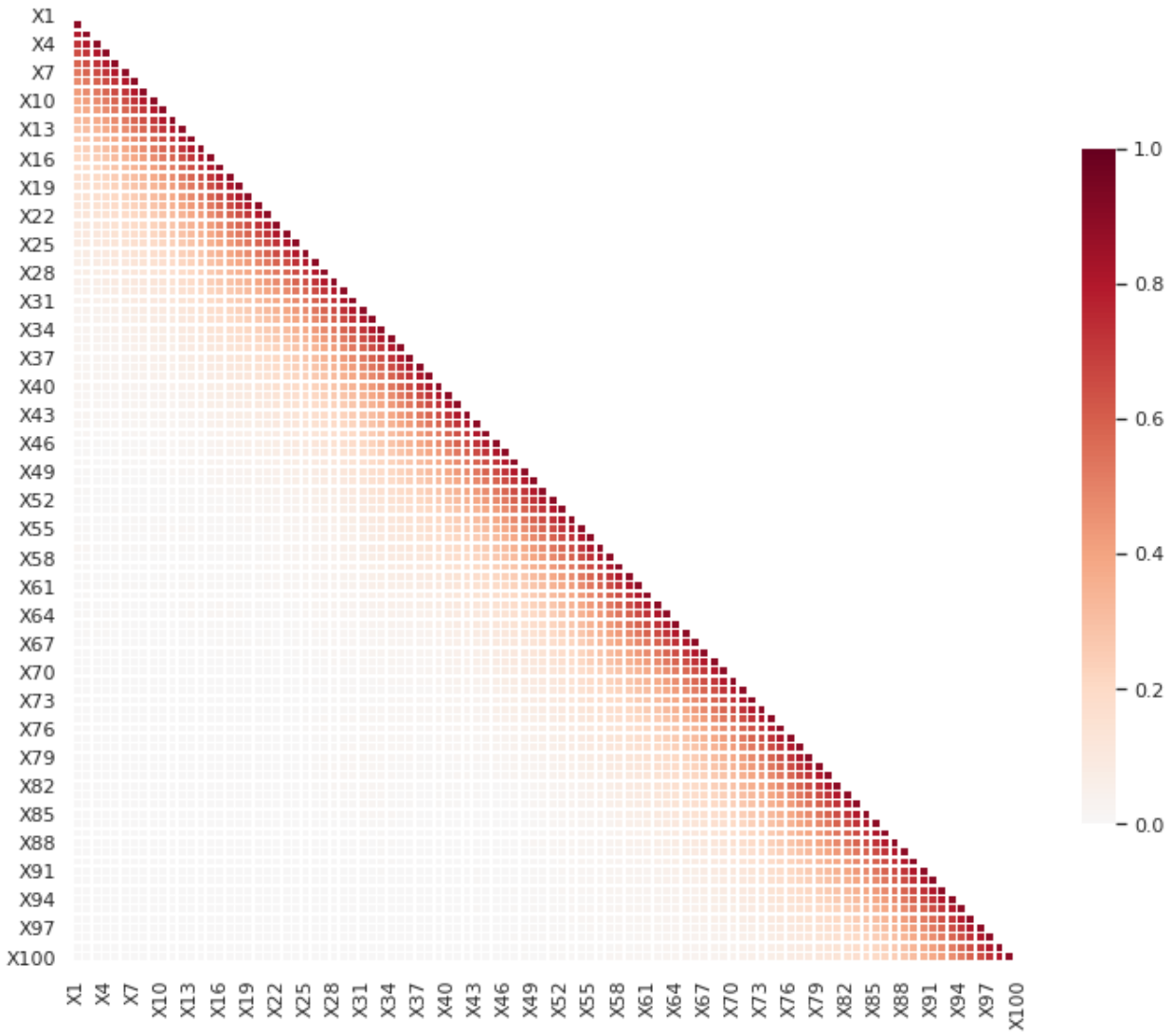
Algorithm:

1. Draw samples $Z^{(i)}$ from $\mathcal{N}(0, \Sigma)$ for $i = 1, \ldots, n$
2. Transform each sample to a uniform by $X_j^{(i)} = \Phi(Z_j^{(i)})$ for each $i = 1, \ldots, n; j = 1, \ldots, K$

where Φ denotes the standard normal cumulative distribution function.

To check wether this works let us simulate a small data set in this fashion. We need to check that the empirical correlation matrix resembles the ones from above, and that the marginal distributions are approximately uniform.

```
n = 25_000
mvnormal = multivariate_normal(cov=Sigma)
Z = mvnormal.rvs(n, random_state=0)

columns = [f"X{k}" for k in range(1, 101)]
XX = norm.cdf(Z)
XX = pd.DataFrame(XX, columns=columns)

shared.correlation_heatmap(XX)
```



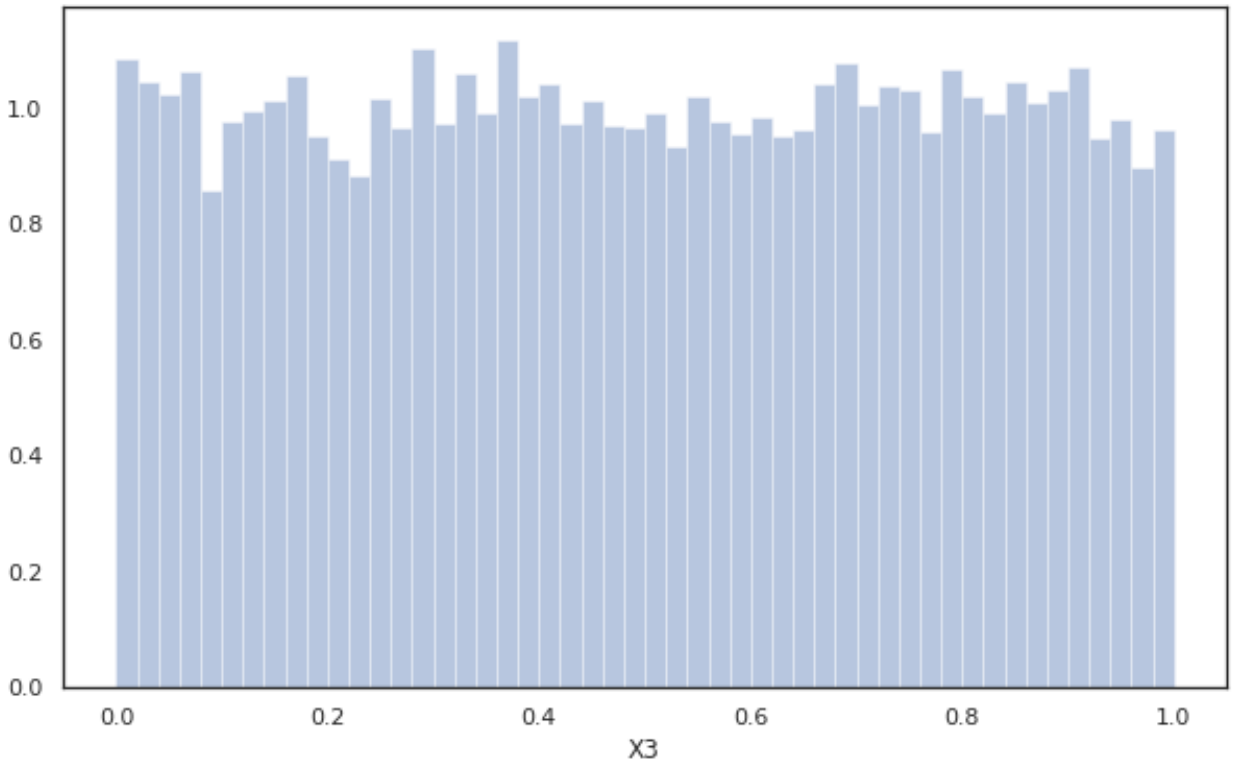
And checking that each column is really uniform distributed by considering a histogram (here we select a few columns randomly since looking at all 100 is too annoying..)

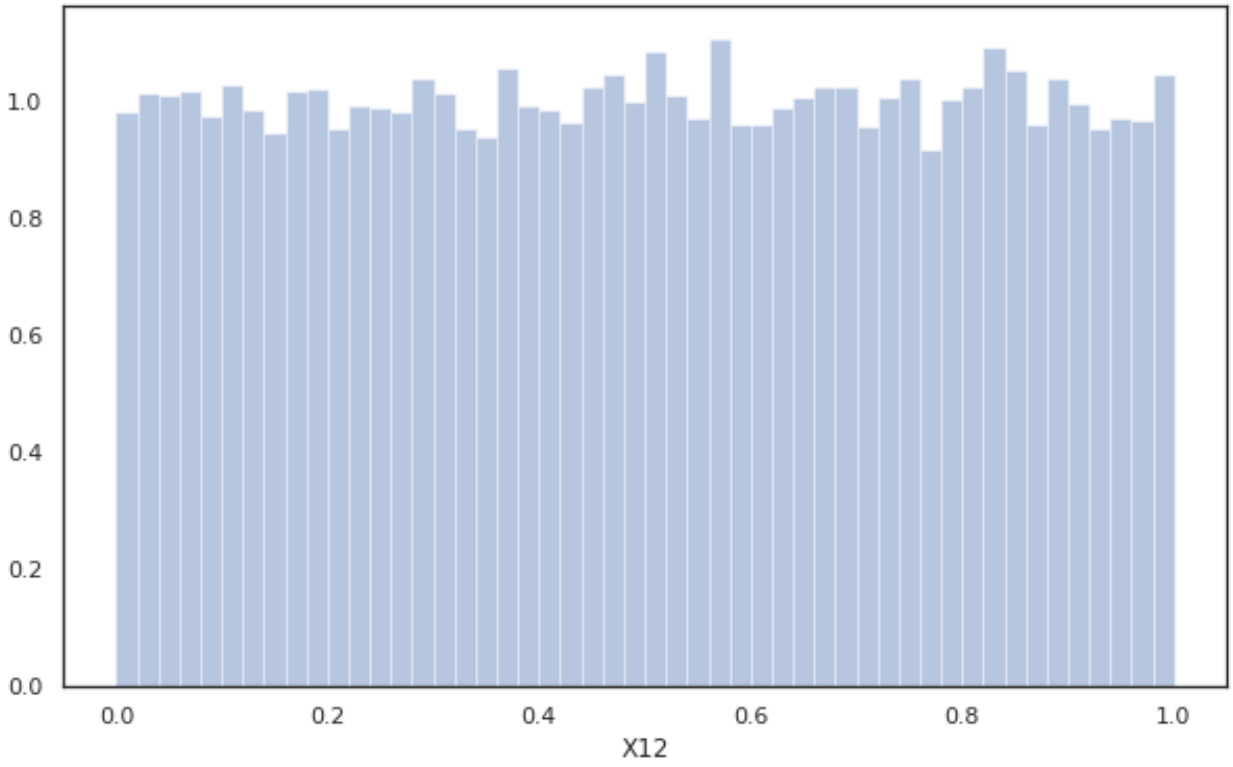
```
for col in ["X3", "X12", "X37"]:
    sns.distplot(
        XX[col],
        bins=50,
        norm_hist=True,
        kde=False,
```

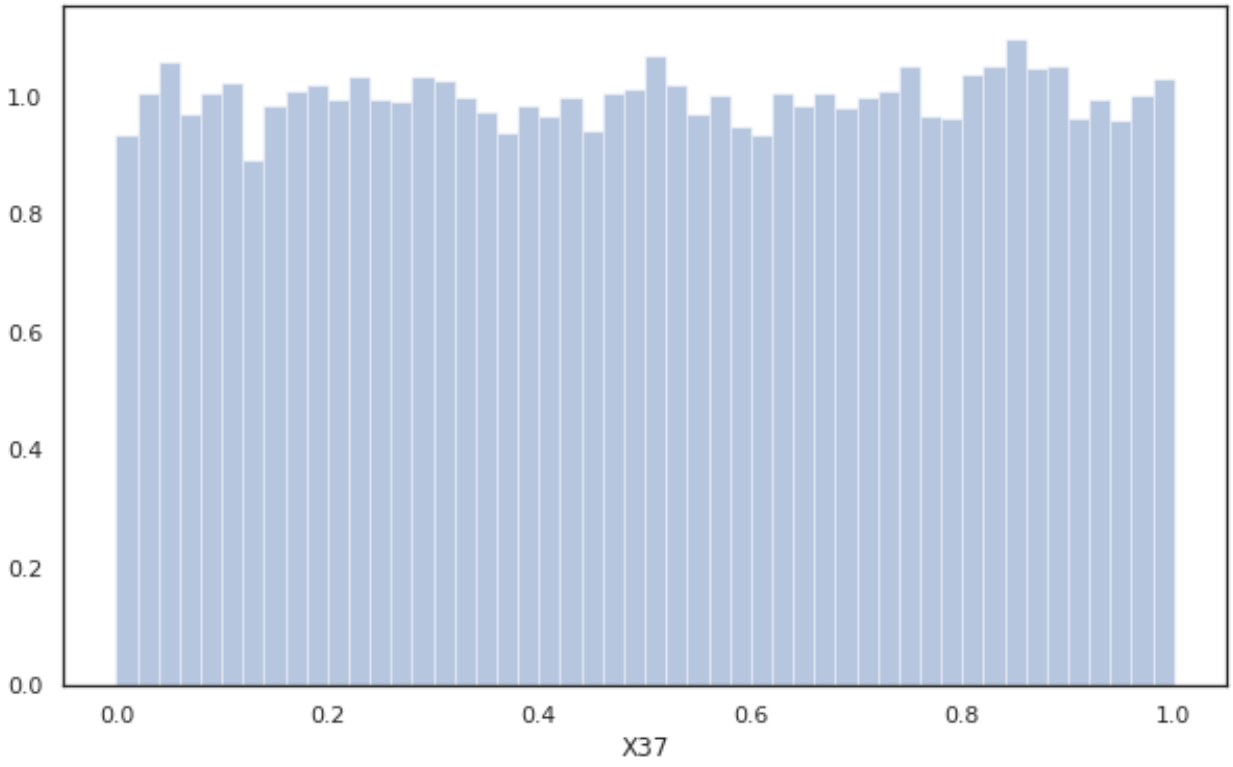
(continues on next page)

(continued from previous page)

```
        hist_kws={"range": [0, 1]}
    )
    plt.show()
```




1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
X12


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
X37

Recap - Features

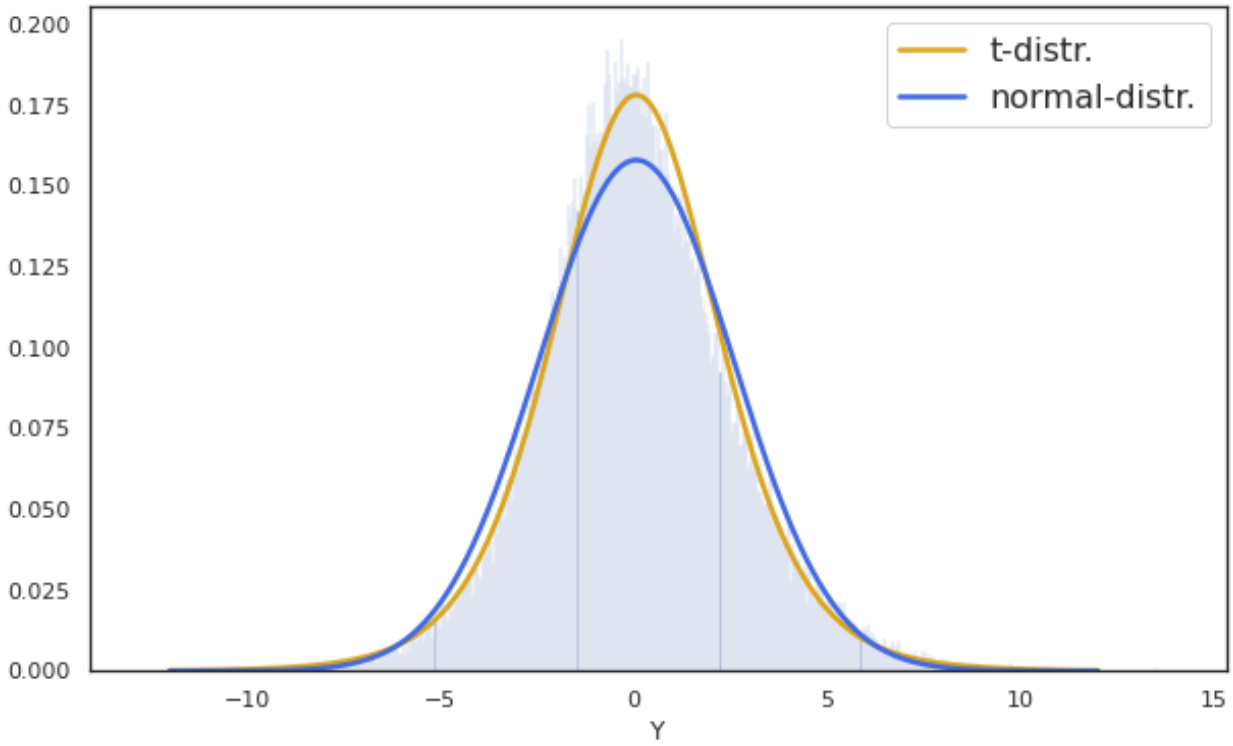
We have seen that the covariance/correlation structure of the features is very similar to a Matern covariance. A major component of feature analysis is to check wether one can reduce the dimensionality of the feature set. As all our features are distributed identically and the correlation structure does not admit any grouping we cannot apply classical techniques such as PCA for sensible dimensionality reduction. This means if we want to reduce the dimensionality we have to actually find and eliminate the features that are unrelated with the outcome or have a negligible effect.

Outcomes

To learn more about the outcome structure I fit the parameters of a normal and t distribution on the outcomes, plot the outcomes using a histogram and draw the two distributions on top. Visualizing the outcome distribution can be important to check wether one needs to transform the distribution, e.g. in the case of positive data.

```
mle_tuple_t = t.fit(y.values.flatten(), floc=y.values.mean())
mle_tuple_norm = norm.fit(y.values.flatten())

t_distr = t(*mle_tuple_t)
norm_distr = norm(*mle_tuple_norm)
```

```
x = np.linspace(-12, 12, 1000)
ax = sns.distplot(y, kde=False, norm_hist=True, bins=500)
ax.plot(x, t_distr.pdf(x), linewidth=2.5, color="goldenrod")
ax.plot(x, norm_distr.pdf(x), linewidth=2.5, color="royalblue")
ax.legend(['t-distr.', 'normal-distr.'], prop={'size': 16})
plt.show()
```



From the plot we can see that the outcomes are more tightly centered than we would expect from a normal distribution. The fitted t-distribution fits the outcome distribution better but still not optimal. Nevertheless if we are willing to

assume exogeneity and an additive error structure, we would expect from the above plot that the model errors follow a more tightly centered distribution with large tails rather than a normal.

The most important insight from the above plot is that there is no need to transform the outcome for a better model fit, as is the case for constrained or positive data.

Relationship Between Outcomes and Features

To better understand how the outcomes are affected by the features I consider one dimensional regression plots. That is, for each feature X_k ($k = 1, \ldots, K$) I consider the plot of Y vs. X_k. (Offline I also considered all two dimensional interaction plot but these are too many to be displayed here and they did not provide any useful insights.)

Since K (=100) plots are quite a few plots I simpy show a video where we iterate through all plots very fast...

The black line seen in the plot denotes the regression line for the model $Y \sim \beta_0 + \beta_1 X_k + \beta_2 X_k^2 + \beta_3 X_k^3$.

```
Video(ROOT / "figures" / "no_interaction.mp4", embed=True, html_attributes="controls␣
↪muted")
```

```
<IPython.core.display.Video object>
```

It is clear that from these univariate effect plots we cannot read-off the entire model, nevertheless these plots contain a great amount of information.

Firstly, we see that "the effect of X_k on Y" is, in a sense, *continuous* in k. That makes sense since we already observed that the features follow a Matern style covariance structure which implies that neighboring features (here: neighboring in the sense that the k's are close) are heavily correlated. Hence, we would expect that these neighboring features *look like* they have a similar effect on the outcome.

Secondly, we see that for many k's the effect looks negligible. In the next steps we analyze this structure of the data more closely and try to establish a more quantitative answer to whether some features are irrelevant.

Lastly, before jumping into a quantitative analysis let me provide a guess on how the above data structure could have emerged. Say we model the outcomes using an additive model and each component is modeled by a 3rd degree polynomial. We then pick a few features, say 2-5, and give them non-zero coefficients. Since the features were created with the special covariance structure from above we would expect that for all features that are close to features with non-zero coefficients, the estimated coefficents would also be non-zero, but shrunk towards zero.

Quantitative Analysis

To get a more quantatitive understanding of which features matter I will

1. Fit a regularized linear model (first-degree, second-degree) using the Lasso and disregard features with zero-coefficient
2. Use a recursive feature elimination with cross-validation
3. Fit a third-degree ridge regression for each feature dimension and compare coefficients

I wont dive too deep into the details of each approach as in the end I decided to pusue a very different route.

Each of the following approaches could have been used to select a small subset of features on which another model could have been fit in a second stage. I did not pursue this strategy in my final model, which is why I do not provide any detailed explainations. Nevertheless the results from below convinced me that the main effects in the data are sparse, which led me to choose hyperparameters in my final machine learning model that perform better under sparsity. This approach also goes well with the "Bet on Sparsity" principle, on which Rob Tibshirani writes:

> Hastie et al. (2001) coined the informal "Bet on Sparsity" principle. The l1 methods assume that the truth is sparse, in some basis. If the assumption holds true, then the parameters can be efficiently estimated using l1 penalties. If the assumption does not hold—so that the truth is dense—then no method will be able to recover the underlying model without a large amount of data per parameter.

The Lasso Approach

To find relevant features we fit a l1 regularized linear model with a) all first-degree terms and b) all first-degree, second-degree and first-degree-interaction terms. For case a) I will also plot the Lasso path. I won't do that for case b) since there will be too many coefficients.

For both approaches I find the *optimal* regularization parameter via 5-fold cross-validation on 50 different alphas.

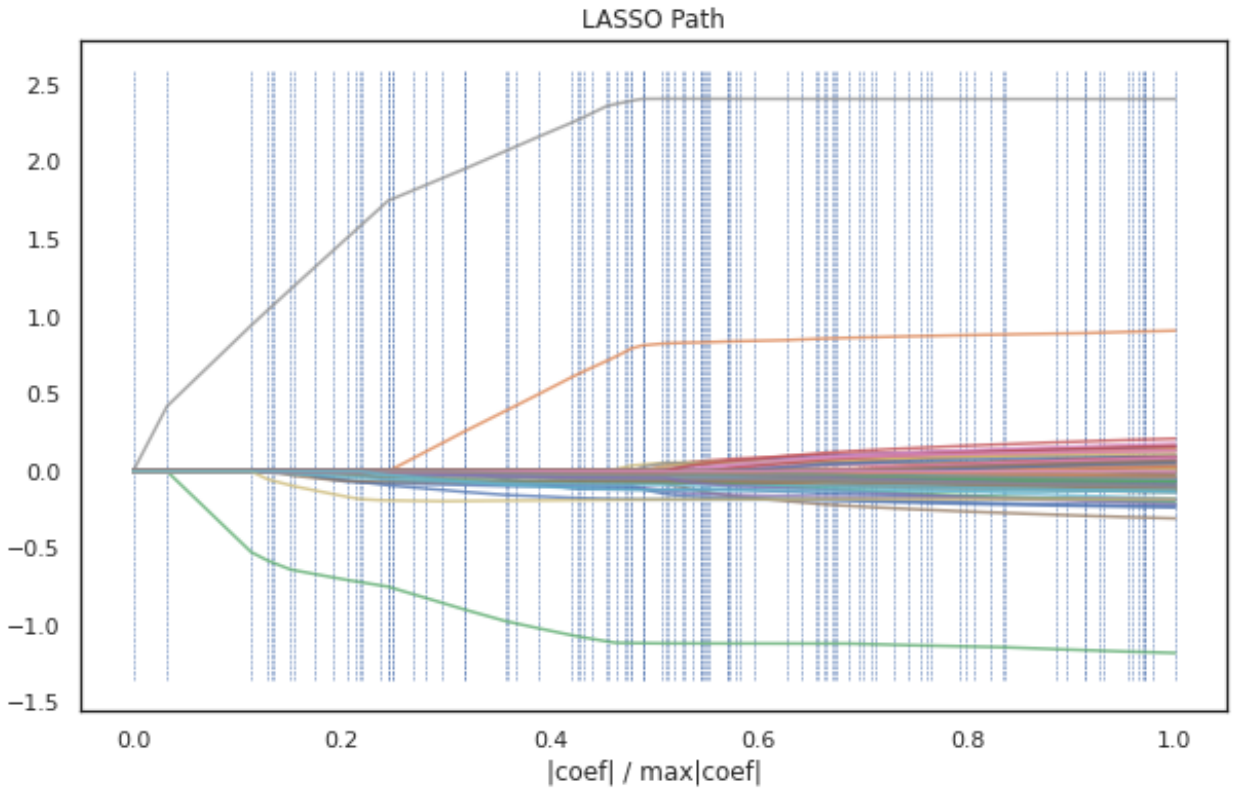
a: first-degree

```
lasso_model = LassoCV(eps=0.05, n_alphas=50, cv=5)
lasso_model = lasso_model.fit(X, y)

relevant = X.columns[lasso_model.coef_ != 0].to_list()
print("Relevant features chosen via Lasso:", relevant)
```

```
Relevant features chosen via Lasso: ['X3', 'X12', 'X14', 'X37', 'X38', 'X39', 'X40']
```

```
_, _, coefs = lars_path(X.values, y.values, method='lasso', verbose=True)

xx = np.sum(np.abs(coefs.T), axis=1)
xx /= xx[-1]

plt.plot(xx, coefs.T, alpha=0.75)
ymin, ymax = plt.ylim()
plt.vlines(xx, ymin, ymax, linestyle='dashed', lw=0.5)
plt.xlabel('|coef| / max|coef|')
plt.title('LASSO Path')
plt.axis('tight')
plt.show()
```

```
.
```



b: second-degree

```
poly = PolynomialFeatures(degree=2, include_bias=False)

XX = poly.fit_transform(X)
poly_columns = [
    col.replace(" ", ":") for col in poly.get_feature_names(X.columns)
]
XX = pd.DataFrame(XX, columns=poly_columns)
```

```
lasso_model_poly = lasso_model.fit(XX, y)

relevant_poly = XX.columns[lasso_model_poly.coef_ != 0].to_list()
print("Relevant features chosen via Lasso:", relevant_poly)
```

```
Relevant features chosen via Lasso: ['X3', 'X5', 'X37', 'X38', 'X57', 'X92', 'X2:X12',
↪ 'X3:X4', 'X3:X5', 'X3:X12', 'X3:X57', 'X5:X9', 'X5:X92', 'X5:X99', 'X7:X12',
↪'X9:X12', 'X12^2', 'X12:X14', 'X38^2', 'X38:X39', 'X38:X40', 'X38:X57', 'X38:X92']
```

Recursive Feature Elimination with CV

As above, I will not dive too deep into the inner workings of the RFECV function as this it not my main model / model selection tool. Here I will apply it on only on the first order effects.

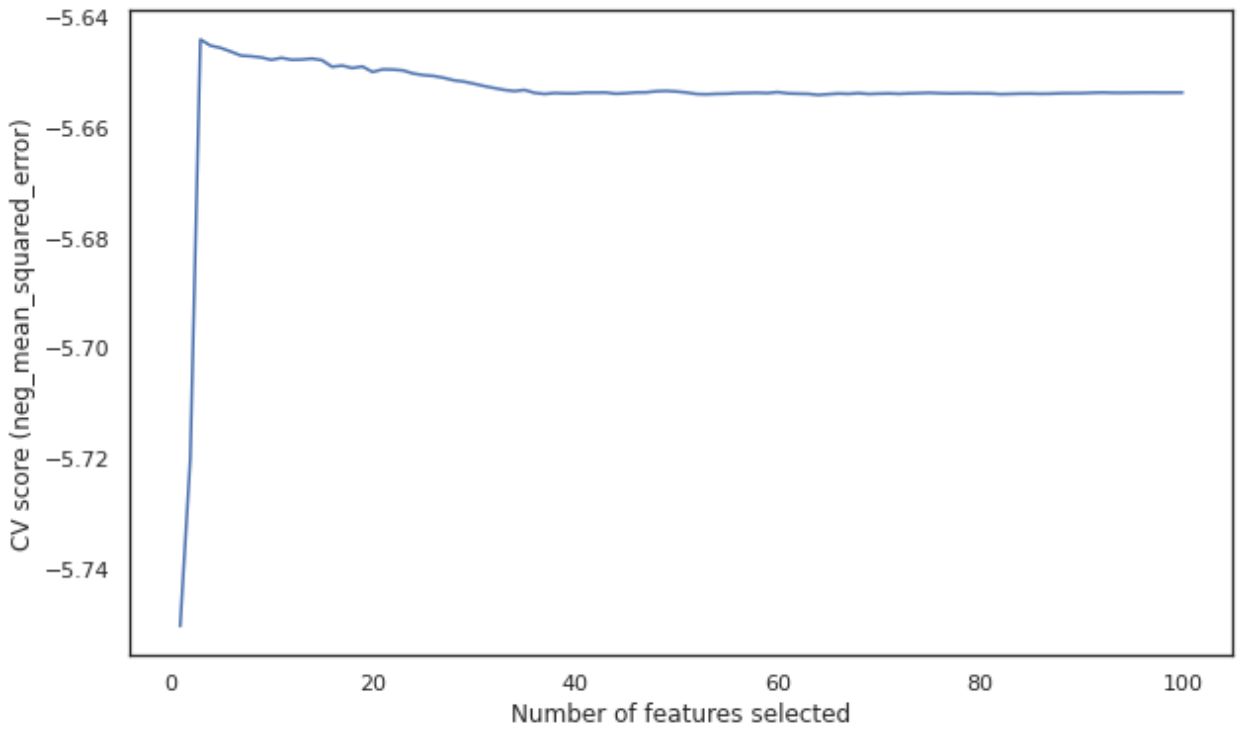
The main idea is to recursively consider smaller and smaller subsets of features. I use 5-fold CV to compute the *optimal* subset and plot the negative mean squared error against the number of features.

Note. As mentioned above I only include the first order effects here since the number of terms explodes when also considering the interactions terms. Still, for each included regressor one could fit a 4th degree polynomial instead of the 1st degree effect I consider below.

```
linear_model = LinearRegression()
rfecv = RFECV(estimator=linear_model, step=1, cv=5, scoring="neg_mean_squared_error")
rfecv.fit(X, y)
print("Optimal subset of features: ", X.columns[rfecv.get_support(True)].to_list())
```

```
Optimal subset of features:  ['X3', 'X12', 'X38']
```

```
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("CV score (neg_mean_squared_error)")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```

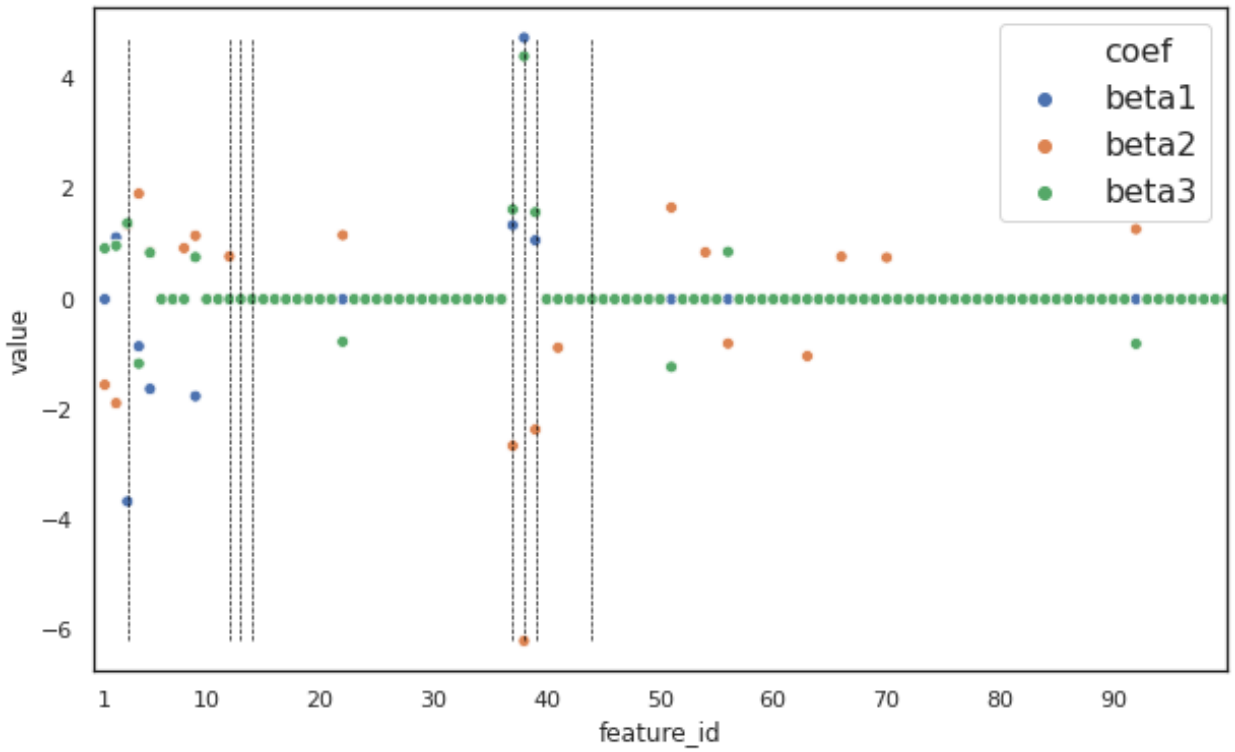


The Ridge Comparison Approach

In this last approach the overall goal is to fit a third-degree linear model with l2 regularization, i.e. a ridge regression, for all features and plot the coefficient values with the feature index on the x-axis. In particular I fit the model $Y \sim \beta_0 + \sum_{k=1}^{K} \beta_{1k} X_k + \beta_{2k} X_k^2 + \beta_{3k} X_k^3$ using an l2 regularization and set all coefficients which are below some threshold (here 0.75) to zero. The results are compared across k. The regularization parameter is chosen via 5-fold cross validation.

```
XX = pd.concat([X ** k for k in range(1, 4)], axis=1)

alphas = np.logspace(-5, 1, num=20)
ridge = RidgeCV(cv=5, alphas=alphas).fit(XX, y)

df_coef = pd.DataFrame(
    ridge.coef_.reshape((-1, 3), order="F"),
    columns=["beta1", "beta2", "beta3"],
    index=range(1, len(X.columns) + 1)
)

threshold = 0.75
df_coef[df_coef.abs() < threshold] = 0
df_coef = df_coef.rename_axis(
    "feature_id", axis=0
).reset_index().melt(id_vars="feature_id", var_name="coef")
df_coef["value"] = df_coef["value"].astype(float)
```

```
plt.figure()
ax = sns.scatterplot(
    x="feature_id", y="value", hue="coef", data=df_coef
)
ymax, ymin = df_coef["value"].max(), df_coef["value"].min()
ax.vlines([3, 12, 13, 14, 37, 38, 39, 44], ymin=ymin, ymax=ymax, color="black", lw=0.
↪65, linestyle="dashed")
plt.legend(prop={'size': 16})
plt.xlim(0, 100)
plt.xticks((1,) + tuple(range(10, 100, 10)))
_ = plt.plot()
```



The dashed black lines represent features that were selected by the first-order Lasso approach

Conclusions (so far)

In all of the methods considered above we have seen that only very few features are selected and accross approaches the same (or at least neighboring) features were selected consistently. This leads me to believe that, as mentioned before, the main effect of the problem is sparse. I would have enjoyed using a simple third-degree polynomial model with subsetted features. Sadly, when compared on the validation set I get a substantially lower prediction error using some of the machine learning methods presented next.

4.1.3 Final

In this last section I will list my top four models and their respective performances on the validation set. Please note that only for my final model I will explain how the fitting procedure works in detail.

I consider a

1. Linear model using a subset of features
2. Deep neural network with dropout regularization
3. Two-stage random forest with feature selection
4. Gradient tree boosting (*the final model*)

If you only care about the final model please jump directly to subsection 4, in which I explain how the model is fit on a theoretical basis as well as how to implement it in Python. Note also that in the very last subsection I include a figure comparing the validation mean squared error for all of the presented models.

Preliminaries

```
import os
from pathlib import Path

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

from catboost import CatBoostRegressor

ROOT = Path(os.getcwd()).parent
np.random.seed(0)
```

```
df_train = pd.read_parquet(ROOT / "bld" / "train_simulated.parquet")
df_val = pd.read_parquet(ROOT / "bld" / "validate_simulated.parquet")

y_train = df_train["Y"]
X_train = df_train.drop("Y", axis=1)

y_val = df_val["Y"]
X_val = df_val.drop("Y", axis=1)
```

1. Linear Model Using a Subset of Features

Here I fit a simple two-step 3rd degree polynomial regression model to benchmark the machine learning methods from below. The procedure has two steps as I only consider the features $\{X_3, X_{12}, X_{38}\}$. This set of features was selected in the previous section using the recursive feature elimination strategy. However, basically all other feature selection strategies considered in the previous section selected a similar set of features.

```
relevant = [f"X{k}" for k in [3, 12, 38]]

XX_train = X_train[relevant]
XX_val = X_val[relevant]

poly = PolynomialFeatures(degree=3, include_bias=False)
XX_train = poly.fit_transform(XX_train)
XX_val = poly.transform(XX_val)

lm = LinearRegression(n_jobs=3)
lm.fit(XX_train, y_train)

prediction = lm.predict(XX_val)
```

(continues on next page)

(continued from previous page)

```
mse_lm = mean_squared_error(y_val, prediction)
print(f"(Linear Model) MSE: {mse_lm}")
```

```
(Linear Model) MSE: 5.361621926612066
```

2. Deep Neural Network with Dropout Regularization

I fit a deep neural network using the popular keras library ([C+15]) which provides an intuitive API for the powerful tensorflow package ([AAB+15]). The neural network architecture is set using the `build_regressor` function. I choose an archtecture with 7 layers. For the first layer I choose 50 hidden nodes, for the second layer 25 hidden nodes and for all remaining layers I choose 10 hidden nodes. Moreover I use the so called ReLu activation function, which has been proven to outperform the classic sigmoid activation function in several ways, see for example [KSH17]. As overfitting is a big problem with deep networks I employ a popular technique called dropout regularization to mitigate this effect, see [SHK+14]. Dropout leads to neurons in a layer being randomly deactivated for a single epoch during the backpropagation. This avoids neighboring neurons developing a strong dependency, which is said to mitigate overfitting.

Note. I decide to use this specific architecture as I "learned" from the previous section that the main effects are sparse in the sense that most likely only very few features are relevant. However, I also realized from playing around with the linear model while watching the validation error that some effects must be non-linear. An architecture which reduces the nodes from the original 100 input dimensions to 50, then 25 and then 10, forces the network to select relevant features. And by using the additional 5 layers the network can potentially find non-linear signals.

Remark. Since the gradient boosted tree presented below performs so well I did not consider many different architectures. I do believe that the neural networks should be able to perform comparably well if a better architecture is chosen.

```
N_COL = X_train.shape[1]
def build_regressor():
    regressor = Sequential()
    # first hidden layer
    regressor.add(Dense(units=50, activation="relu", input_dim=N_COL))
    regressor.add(Dropout(0.2))

    # second hidden layer
    regressor.add(Dense(units=25, activation="relu"))
    regressor.add(Dropout(0.2))

    # third to tenth hidden layer
    for _ in range(5):
        regressor.add(Dense(units=10, activation="relu"))

    # output layer
    regressor.add(Dense(units=1, activation="linear"))

    # compile model
    regressor.compile(optimizer="adam", loss="mean_squared_error")
    return regressor
```

```
nnet = KerasRegressor(
    build_fn=build_regressor, batch_size=128, epochs=200, verbose=0
)
nnet.fit(X_train, y_train)
```

(continues on next page)

(continued from previous page)

```
prediction = nnet.predict(X_val)
mse_nnet = mean_squared_error(y_val, prediction)
print(f"(Neural Network) MSE: {mse_nnet}")
```

```
(Neural Network) MSE: 5.289789777325298
```

3. Two-stage Random Forest with Feature Selection

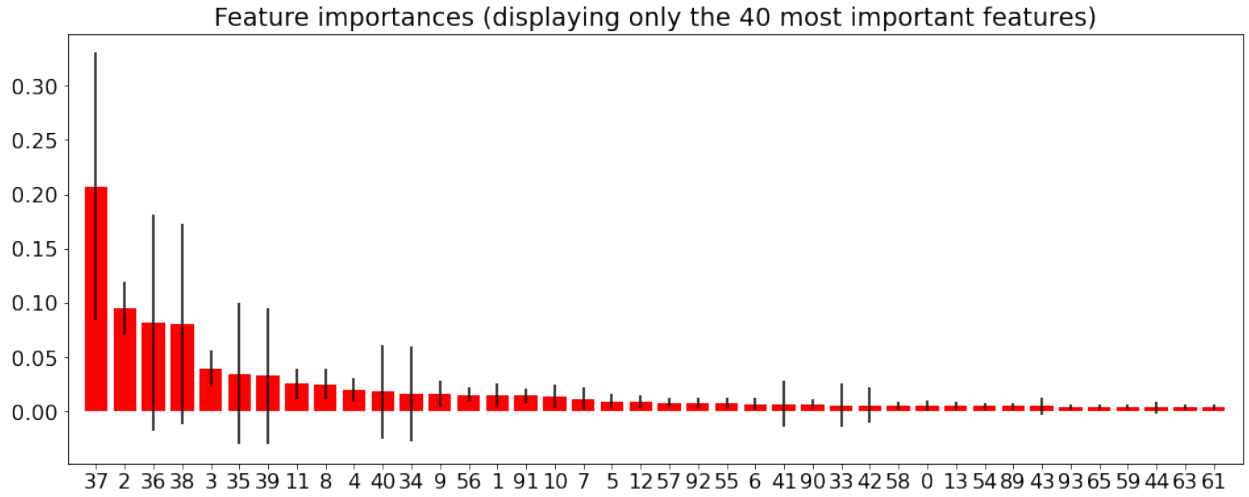
In this two stage procedure I first fit a random forest on the full set of features. I then consider the standard feature importance measure of random forests, which is automatically calculated from the fitting procedure. Using this I select the 30 *most* important features and fit another random forest on this subset of features.

First Stage

```
rf = RandomForestRegressor(
    n_estimators=250,
    max_features=25,
    max_depth=15,
    min_samples_leaf=100,
    bootstrap=True,
    n_jobs=3,
    random_state=1,
)
rf.fit(X_train, y_train.values)

std = np.std(
    [tree.feature_importances_ for tree in rf.estimators_],
    axis=0
)
indices = np.argsort(rf.feature_importances_)[::-1]
relevant = [f"X{i+1}" for i in indices[:30]]

prediction = rf.predict(X_val)
mse_first_stage = mean_squared_error(y_val, prediction)
print(f"First stage MSE: {mse_first_stage}")
```

```
First stage MSE: 5.1615609346407565
```

```
# plotting code: can be safely ignored
plt.rcParams.update({'font.size': 16, 'figure.figsize': (16, 6)})
cut = 40
x = range(X_train.shape[1])[:cut]
y = rf.feature_importances_[indices][:cut]
plt.figure()
plt.title(f"Feature importances (displaying only the {cut} most important features)")
plt.bar(x, y, color="r", yerr=std[indices][:cut], align="center")
plt.xticks(x, indices[:cut])
plt.xlim([-1, cut])
plt.show()
```



Second Stage

```
XX_train = X_train[relevant]
XX_val = X_val[relevant]

rf = RandomForestRegressor(
    n_estimators=250,
    max_features=15,
    max_depth=15,
    min_samples_leaf=100,
    bootstrap=True,
    n_jobs=3,
    random_state=1,
)
rf.fit(XX_train, y_train.values)

prediction = rf.predict(XX_val)
mse_rf = mean_squared_error(y_val, prediction)
print(f"(Random Forest) MSE: {mse_rf}")
```

```
(Random Forest) MSE: 5.120511556417796
```

4. Gradient tree boosting (*the final model*)

The final model I decided to use is a specific variant of a gradient boosted tree. The key concepts of this method are equivalent to the algorithm proposed in the seminal paper by Friedman, see [Fri02]. The version I am using is implemented in the catboost package, which differs slightly from common other implementations. Next I will introduce the theoretical concept of gradient boosting with a particular focus on tree weak-learners. Afterwards I show how to fit a model using `catboost`.

Note. My final prediction submissions are made with the exact model specification as presented in below, but using the complete training data. The corresponding script can be found here final_prediction.py.

Theory

I will first explain the general idea behind boosting and gradient boosting. Then I show how the general formulas simplify when using (regression) trees as base-learners. At last I illustrate how out-of-sample prediction loss may be improved by using ideas from stochastic gradient descent and regularization.

Note. My notation and explaination is guided by [Fri02] and [HTF01].

Notation and Preliminaries

Assume we are given a data set $\{(x_i, y_i) : i = 1, \dots, N\}$, with $x_i \in \mathbb{R}^p$ and $y_i \in \mathbb{R}$. These observations are assumed to be i.i.d. according to some joint distribution $\mathbb{P}_{xy}$. An important goal of statistical learning is to find a function $f^* : \mathbb{R}^p \to \mathbb{R}$ such that

\begin{align*} f^* = \underset{f}{\text{argmin}} , \mathbb{E}^{xy} \left [L(y, f(x)) \right], \end{align*} given some loss function L, where the expectation is taken over the joint distribution of x and y values.

As is very common in statistical learning, boosting is a procedure to approximate f^* by an additive model of the form

\begin{align*} f(x) = \sum_{m=0}^M \beta_m b(x; \gamma_m), \end{align*} where the β_m denote expansion coefficients and $b(\cdot, \gamma) : \mathbb{R}^p \to \mathbb{R}$ denote the so called *base-learners* which are parameterized by γ. If feasible in general we would like to estimate the parameters by solving

\begin{align*} \underset{\{\beta_m, \gamma_m}{*I^M}{\min} \sum{i=1}^N L\left(y_i, \sum_{m=0}^M \beta_m b(x_i; \gamma_m)\right). \end{align*}

For a general loss function and base-learner, however, this optimization is computationally intractable. We will also see that there are other reasons why we would like to estimate the coefficients in a different fashion.

General Concept

A simple algorithm to approximate the coefficient estimates from above is to fit each tuple (β_m, γ_m) in a stage-wise fashion. That is, one starts with an initial guess f_0 and then

for each $m = 1, \dots, M$ do:

1. $(\beta_m, \gamma_m) = \underset{\beta, \gamma}{\text{argmin}} \sum_{i=1}^N L\left(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)\right)$
2. $f_m(x) = f_{m-1}(x) + \beta_m b(x, \gamma_m)$.

This simplifies the optimization problem from above consideribly, but again, for general loss functions and base-learners step 1 can be hard to solve.

Gradient Boosting

Gradient boosting, as proposed in [Fri00], approximately solves the first step from above for differentiable loss functions using a three step procedure. In the m-th loop from above we do

1. $r_{im} := -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$
2. $\gamma_m = \underset{\gamma, \rho}{\text{argmin}} \sum_{i=1}^N \left(r_{im} - \rho b(x_i; \gamma)\right)^2$
3. $\beta_m = \underset{\beta}{\text{argmin}} \sum_{i=1}^N L\left(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma_m)\right)$.

That is, we first compute the gradient, whose entries we call *pseudo-residuals*, and then we fit a single base-learner using the least-squares method. Lastly we have to solve a general optimization problem, but only in a single variable, which can be done efficiently with state-of-the-art optimization algorithms.

With squared-error loss ($L(y,x) = \frac{1}{2}(y-x)^2$) the pseudo-residuals r_{im} are equivalent to the actual residuals $r_{im} = y_i - f_{m-1}(x_i)$. This allows for the nice interpretation of the procedure that given any step m, we consider the deviations of the labels y_i from our current estimate $f_{m-1}(x_i)$ and train a model to fit these deviations. I.e. in each iterations the model tries to improve the fit in the regions where the fit is worst. With other loss functions, say absolute-error loss ($L(y,x) = |y-x|$) we get $r_{im} = \text{sign}(y_i - f_{m-1}(x_i))$ and in the m-th step the base-learner is fit to simply predict the direction to which a sample deviates, which is more robust to outliers as it ignores the magnitude of the deviation.

Gradient Tree Boosting

A popular choice of base-learners are decision trees. When using trees as base learners some steps in the generic algorithm simplify and some can even be improved. First, let us formally define a tree. A regression tree with J terminal-nodes is a function

\begin{align*} T\left(x; {\alpha_j, R_j}*{j=1}^J *right) = *sum{j=1}^J \alpha_j \mathbb{1}*{R{j}}(x), \end{align*} where $\{R_j\}_{j=1}^J$ is a partition of the feature space $\mathbb{R}^p$. Strictly speaking, J is also a parameter, however, it is usually considered a *hyper-parameter* which has to be chosen using prior information or via methods like cross-validation.

Let us now consider the second step of the above algorithm. First note that optimizing over ρ is irrelevant in the case of trees as we can always define $\tilde{\alpha}_j := \rho\alpha_j$. But then solving 2 is equivalent to solving

\begin{align*} {\alpha_{jm}, R_{jm}}*{j=1}^J =: *gamma_m = *underset{*gamma}{*text{argmin}} , *sum{i=1}^N \left(r_{im} - T(x_i; \gamma) \right)^2 \end{align*} For a given J this combinatorial optimization problem is again computationally intractable, but there are many algorithms that can approximate the solution; see for example the CART algorithm in [HTF01].

Henceforth say we have (approximately) solved the tree optimization problem (step 2) and are left to optimize for the constant β_m (step 3). As will be seen, with trees we can even go one step further and choose an optimal value for each terminal-node region. Let $\gamma_m = \{\alpha_j, R_j\}_{j=1}^J$ be the fitted parameters from step 2. The next simplification when using trees stems from the fact the the R_j's form a partition of the feature space. We can rewrite the sum over the individuals as a sum over the terminal-node regions, as a tree predicts the same value for each region. That is, step 3 becomes

\begin{align*} {\beta_{jm}}*{j=1}^J = *underset{*beta_1,*dots,*beta_J}{*text{argmin}} , *sum{j=1}^J \sum_{x_i \in R_j} L \left(y_i, f_{m-1}(x_i) + \beta_j \right) ,, \end{align*} which can be solved for each region seperately. Note that we can write $L\left(y_i, f_{m-1}(x_i) + \beta_j\right)$ instead of $L\left(y_i, f_{m-1}(x_i) + \beta_j\alpha_j\right)$.

As an example, with squared error loss we would then get $\beta_{jm} = \text{mean}(y_i - f_{m-1}(x_i) : x_i \in R_{jm}) = \text{mean}(r_{im} : x_i \in R_{jm})$.

Algorithm

For the sake of clarity I illustrate the complete gradient tree boosting algorithm. This corresponds to Algorithm 10.3 (Gradient Tree Boosting Algorithm) in [HTF01] with minor modifications.

Input: M (number of trees), J (number of terminal nodes), L (loss function), $\{(x_i, y_i)\}_{i=1}^N$ (training data).

1. $f_0 = \underset{\gamma}{\text{argmin}} \sum_{i=1}^N L(y_i, \gamma)$
2. For $m = 1, \ldots, M$:
   - a) For $i = 1, \ldots, N$ compute $r_{im} := -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$
   - b) Fit a tree to $\{(x_i, r_{im})\}_{i=1}^N$ resulting in regions $\{R_{jm}\}_{j=1}^J$

- c) For $j = 1, \dots, J$ solve $\gamma_{jm} = \underset{\gamma}{\operatorname{argmin}} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$
- d) Update $f_m = f_{m-1} + \sum_{j=1}^{J} \gamma_{jm} \mathbb{1}_{R_{jm}}$

3. Return $f = f_M$

Enhancements

Until now we have not mentioned two key concepts used widely in the statistical and machine learning literature. *Injecting randomness*, which is used to decorrelate trees and therefore to avoid overfitting, and *regularization*, which is is also used to avoid overfitting.

- ***Injecting Randomness.*** From empirical experience it has been seen that methods like bagging [Bre96] and random forests [Bre01] outperform similar algorithms that do not explicitly make use of randomness. Similarly stochastic gradient descent is considered superior to the classical gradient descent algorithm when used in a machine learning setting; see [BCN16]. The easiest way to inject randomness to the gradient boosting algorithm is described in [Fri02]. In the second step of the above algorithm we simply consider a randommly shuffled subset of the data of size $N' < N$. On top of potentially improving out-of-sample fit, this can lead to significantly shorter training times.
- ***Regularization.*** Similarly as injecting randomness, a popular technique to avoid overfitting is regularization. Especially shrinkage methods like ridge regression and lasso have gained immense popularity; see for example [HTW15]. An immediate enhancement to the above algorithm is to include a shrinkage paramater $0 < \nu \leq 1$ in the updating step, which is usually called the *learning rate*. In step 2.d) we then modify the equation slightly to get $f_m = f_{m-1} + \nu \sum_{j=1}^{J} \gamma_{jm} \mathbb{1}_{R_{jm}}$. It was found empirically that small values of the learning rate, $\nu < 0.1$, lead to better generalization; see [Fri00].

Catboost

Catboost is an open-source library for gradient boosting on decision trees. Its implementation differs slighty from the above algorithm. As of right now it outperforms or ties with most other open-source boosting libraries such as LightGBM, XGBoost and H2O; see the benchmarks here. Next I will list the main differences and refer to the respective papers for reference.

- ***Categorical Features.*** The main new feature in `catboost` is the clever support of categorical features. Decision trees cannot usally deal with categorical features with more than 2 states. (This is of course because the binary relation "$\leq$" must not be defined on the discrete space.) A common approach is the so called *one-hot* encoding, where we introduce a new binary feature for the activation of each state. If the feature set includes many categorical features and the state space is large for many, then one-hot encoding can blow up the dimensionality of the problem to a problematic extent. Another approach is to use target statistics, in which we try to map the categorical features to numerical ones. Consider a categorical feature and let it be denoted by the k-th feature. Frequently the target statistic is chosen as to approximate the conditional mean, i.e. $\tilde{x}_{ik} \approx \mathbb{E}_{xy}[y \mid x = x_{ik}]$. See [DEG18] for details.
- ***Unbiased Gradients.*** It is argued in [DGG+17] that gradient boosting suffers from a bias generated by using biased gradient estimates, which leads to overfitting. Theorem 1 in [PGV+17] proves this result under some conditions on the algorithm. A solution to avoid this effect is proposed in the above papers and implemented in `catboost`.
- ***Oblivious Trees.*** The base-learner used in `catboost` is not a regular decision tree but an *oblivious tree*, sometimes also called a *decision table*. The main difference to regular decision trees is that all nodes on a given level have to split on the same feature and the same point. This is why oblivious trees are usually longer than standard trees since they need more levels to capture non-linearities. Recently this form of tree structure has captured attention in the machine learning community and is being used often together with gradient boosting machines; see for example [LO17], [FModry16] and [PMB19]. For a general definition of oblivious trees see [KL95].

Final Remark on the Theory.

In the above I have talked about the (historical) development of the gradient tree boosting algorithm, however, I have not explained why boosting achieves the goal of function approximation so well. Given the standard squared-error loss it is known that the function f^* which minimizes the expected loss is the conditional expectation $f^*(x) = \mathbb{E}_{xy}[y \mid x = x]$. Do we have any guarantee that boosting is at least point wise consistent then? Specific variants of the boosting algorithm and consistency thereof have been of interest in the recent literature; see [BC17] or [LKPM19]. For the very first boosting algorithm (which was actually used for classification) [SPH07] show consistency under regularity conditions. At last [ZY05] consider consistency of the boosting algorithm under early-stopping.

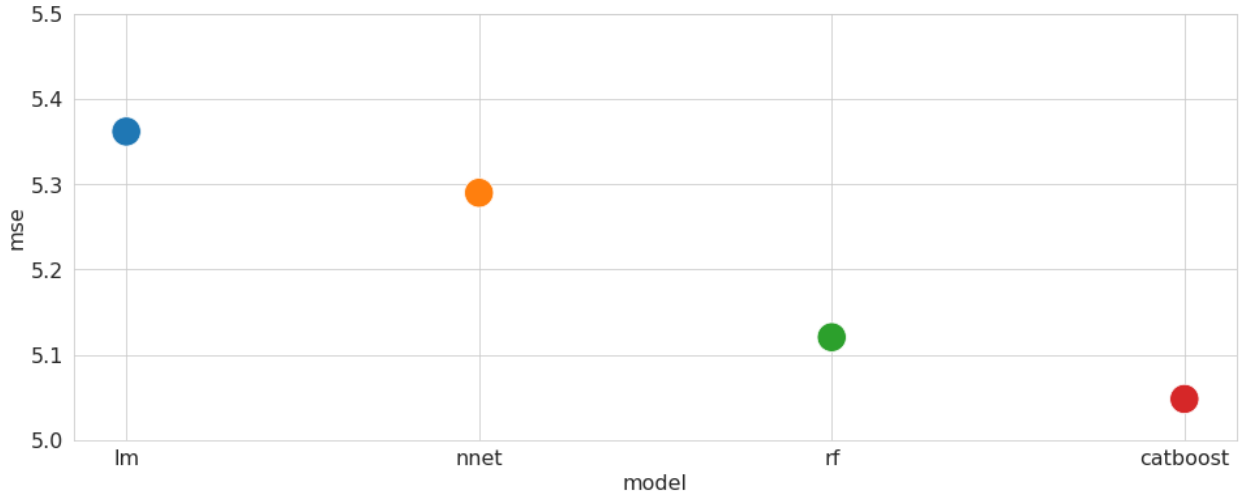
Implementation / Application

```
gbt = CatBoostRegressor(
    iterations=1500,
    learning_rate=0.01,
    depth=5,
    loss_function="RMSE",
    random_state=1,
)
gbt.fit(X_train, y_train, verbose=False)

prediction = gbt.predict(X_val)
mse_gbt = mean_squared_error(y_val, prediction)
print(f"(Catboost) MSE: {mse_gbt}")
```

```
(Catboost) MSE: 5.048105271497743
```

MSE Comparison

```
models = ["lm", "nnet", "rf", "catboost"]
mses = [mse_lm, mse_nnet, mse_rf, mse_gbt]
data = pd.DataFrame(zip(models, mses), columns=["model", "mse"])

sns.set_style("whitegrid")
plt.ylim(5, 5.5)
_ = sns.scatterplot(
    x="model", y="mse", data=data, hue="model", s=500, legend=None
)
```



4.2 Stock Data

In the following sections I present my investigation of the stock data set. Again, I start with a quick exploration of the data set as well as a short explaination of how I chose to split the data into training and validation samples. Then I jump directly to the last section where I present my final model and some benchmarks.

4.2.1 Data Description

In the following I will present my model for the stock data set. Before I could fit my final model I had to transform the data. Next I will discuss how I changed the data structure and how I chose split the data intro training and validation parts.

```
import os
from pathlib import Path
import pandas as pd

ROOT = Path(os.getcwd()).parent
```

```
df = pd.read_parquet(ROOT / "data" / "stock_data.parquet")
df.iloc[:5, :10]
```

```
         date         Y      a2me       aoa        at    at_adj       ato  \
0  1965-01-31  0.461364  0.513089  0.282723  0.787958  0.319372  0.172775
1  1965-01-31  0.542868  0.240838  0.774869  0.939791  0.340314  0.183246
2  1965-01-31  0.249849  0.633508  0.096859  0.222513  0.884817  0.785340
3  1965-01-31  0.371568  0.439791  0.463351  0.903141  0.434555  0.112565
4  1965-01-31 -0.177803  0.654450  0.335079  0.704188  0.958115  0.848168

       beme  beme_adj      beta
0  0.484293  0.609948  0.335079
1  0.232984  0.308901  0.526178
2  0.774869  0.787958  0.853403
3  0.494764  0.643979  0.570681
4  0.549738  0.793194  0.866492
```

The original data set contains 1_629_155 observations of stock returns including 63 features. One of these features which is of particular importance is the date. In comparison to classical panel data, however, the above data does not have a unit index. That is, we cannot know which units move between time periods. Observations are measured from the 01.31.1965 until the 31.05.2014. Again, as in the simulated case, testing observations are marked with a `NaN` in the outcome column. Here the testing observations are all observations starting from the 31.01.2004 until the last observed time period.

Cleaning the Data

Before training my models I cleaned the data in several ways. First, I transformed the date column to a year column and a one-hot-encoded quarter column. I.e., `data = 1965-01-31` becomes `year = 1965` and all dummies will be zero, as the first quarter is integrated in the intercept. I then dropped all observations older than 1990. I did this since I believed that the any information in the data of the '70s-'90s which could be used to explain stock returns was unlikely to still explain modern stock returns. Also I wanted to reduce the size of the data set. At last I dropped all observations which had absolute returns greater than 6, as from looking at a fine histogram, these seemed to be outliers. The data cleaning script can be found here clean_data.py.

Train / Validation Split

As I did not want to ignore the time dimension for the train / validation split I constructed the respective sets as follows. I grouped the cleaned data set into sets by year. For each year I split the respective set into 80% training and 20% validation set. Lastly I concatened the smaller sets together to form the final training and validation sets. Using this strategy I can train my model on all time-periods and evaluate the performance on all time-periods. The specific implementation is given in the script train_test_split.py.

4.2.2 Final

My final prediction model is build on the transformed data set from the previous section. In the following I consider three models.

1. Linear model
2. Two-stage linear model (*the final model*)

Again, if you only care about the final model please jump directly to subsection 2.

Preliminaries

```
import os
from pathlib import Path

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LassoCV
from sklearn.metrics import mean_squared_error

ROOT = Path(os.getcwd()).parent
```

```
df_train = pd.read_parquet(ROOT / "bld" / "train_stock.parquet")
df_val = pd.read_parquet(ROOT / "bld" / "validate_stock.parquet")

y_train = df_train["Y"]
X_train = df_train.drop("Y", axis=1)

y_val = df_val["Y"]
X_val = df_val.drop("Y", axis=1)
```

1. Linear Model

Here I fit a simple unregularized linear model which is used as a lower benchmark.

```
lm = LinearRegression()
lm.fit(X_train, y_train)

prediction = lm.predict(X_val)
mse_lm = mean_squared_error(y_val, prediction)
print(f"(Linear Model) MSE: {mse_lm}")
```

```
(Linear Model) MSE: 0.9494373195140412
```

2. Two-stage Linear Model (*final model*)

To mix things up, here I select features using a Lasso approach. With these features I then fit a simple 2nd degree polynomial model. The code which I used to construct the final predictions can be found in the script final_prediction.py.

Lasso feature selection

The regularization parameter is selected via a 5-fold cross-validation procedure over a logspace grid (a sequence which is linear on a logarithmic scale). I select all columns which have nonzero coefficients.

```
lasso_model = LassoCV(alphas=np.logspace(-2.5, 1, 50), cv=5)
lasso_model = lasso_model.fit(X_train, y_train)

relevant = X_train.columns[lasso_model.coef_ != 0].to_list()
print("Relevant features chosen via Lasso:")
print(relevant)
```

```
Relevant features chosen via Lasso:
['at_adj', 'beme', 'cum_return_12_2', 'cum_return_12_7', 'cum_return_1_0', 'cum_
↪return_36_13', 'd_so', 'e2p', 'free_cf', 'noa', 'pcm', 'pm', 'pm_adj', 'ret_max',
↪'suv', 'year', 'quarter3', 'quarter4']
```

Polynomial regression on subset

```
def make_features(X, relevant_columns):
    """Return 2nd degree polynomial features plus third power."""
    poly = PolynomialFeatures(degree=2, include_bias=False)
    XX = poly.fit_transform(X[relevant_columns])
    XX = np.concatenate((XX, X ** 3), axis=1)
    return XX
```

```
XX_train = make_features(X_train, relevant)
XX_val = make_features(X_val, relevant)

pm = LinearRegression()
pm.fit(XX_train, y_train)
del XX_train # memory ...
```

```
predictions = pm.predict(XX_val)
mse_pm = mean_squared_error(y_val, predictions)
print(f"(Lasso Polynomial Model) MSE: {mse_pm}")
```

```
(Lasso Polynomial Model) MSE: 0.94026689736296
```

4.3 Bibliography

BIBLIOGRAPHY

[AAB+15] Mart'ın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: large-scale machine learning on heterogeneous systems. 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[BC17] Gérard Biau and Benoît Cadre. Optimization by gradient boosting. 2017. arXiv:1707.05023.

[BCN16] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. 2016. quantization overview. URL: https://arxiv.org/abs/1606.04838, arXiv:arXiv:1606.04838.

[Bre96] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. URL: http://dx.doi.org/10.1023/A%3A1010933404324, doi:10.1023/A:1010933404324.

[C+15] Francois Chollet and others. Keras. 2015. URL: https://github.com/fchollet/keras.

[DEG18] Anna Veronika Dorogush, V. Ershov, and A. Gulin. Catboost: gradient boosting with categorical features support. *ArXiv*, 2018.

[DGG+17] Anna Veronika Dorogush, Andrey Gulin, Gleb Gusev, Nikita Kazeev, Liudmila Ostroumova, and Aleksandr Vorobev. Fighting biases with dynamic boosting. *ArXiv*, 2017.

[FModry16] Michal Ferov and Marek Modrý. Enhancing lambdamart using oblivious trees. *ArXiv*, 2016.

[Fri00] Jerome H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[Fri02] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367 – 378, 2002. Nonlinear Methods and Data Mining. URL: http://www.sciencedirect.com/science/article/pii/S0167947301000652, doi:https://doi.org/10.1016/S0167-9473(01)00065-2.

[HTF01] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[HTW15] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. *Statistical Learning with Sparsity: The Lasso and Generalizations*. Chapman & Hall/CRC, 2015. ISBN 1498712169.

[KL95] Ron Kohavi and Chia-Hsin Li. Oblivious decision trees graphs and top down pruning. 1995.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. URL: https://doi.org/10.1145/3065386, doi:10.1145/3065386.

[LO17] Yin Lou and Mikhail Obukhov. Bdt: gradient boosted decision tables for high accuracy and scoring efficiency. 2017. URL: https://doi.org/10.1145/3097983.3098175, doi:10.1145/3097983.3098175.

[LKPM19] Haihao Lu, Sai Praneeth Karimireddy, Natalia Ponomareva, and Vahab Mirrokni. Accelerating gradient boosting machine. 2019. arXiv:1903.08708.

[PMB19] Sergei Popov, Stanislav Morozov, and Artem Babenko. Neural oblivious decision ensembles for deep learning on tabular data. 2019. arXiv:1909.06312.

[PGV+17] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. 2017. arXiv:1706.09516.

[SPH07] B. Schölkopf, J. Platt, and T. Hofmann. Adaboost is consistent. 2007.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[ZY05] Tong Zhang and Bin Yu. Boosting with early stopping: convergence and consistency. *The Annals of Statistics*, 33(4):1538–1579, Aug 2005. URL: http://dx.doi.org/10.1214/009053605000000255, doi:10.1214/009053605000000255.