Contents

Introduction

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.

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.

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.

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.

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)

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.

Data Description

In the following I will present my model for the simulated data set. Before I present the final model I will talk about how I reached the decision and compared the model to other potential candidates.

```
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	X7	
0	2.125298	0.711338	0.683167	0.493706	0.317948	0.374013	0.600761	0.630743	0
1	-1.102707	0.666355	0.422247	0.366055	0.390193	0.499254	0.602870	0.379105	0
2	-2.847834	0.065469	0.307784	0.225924	0.233217	0.528559	0.468785	0.787321	0
3	-0.386088	0.715237	0.487894	0.716503	0.595477	0.619713	0.761047	0.879549	0
4	2.518977	0.144925	0.368206	0.335244	0.426445	0.564264	0.478321	0.466601	0

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 given a `NaN` in the outcome column.

Train / Validation Split

The remaining 80_000 *labelled* data points were (randomly) split further by me 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 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 description of the models I considered then feel free to skip the next section.

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. Note also that in this section we only work with the pure training data.

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()

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")
```

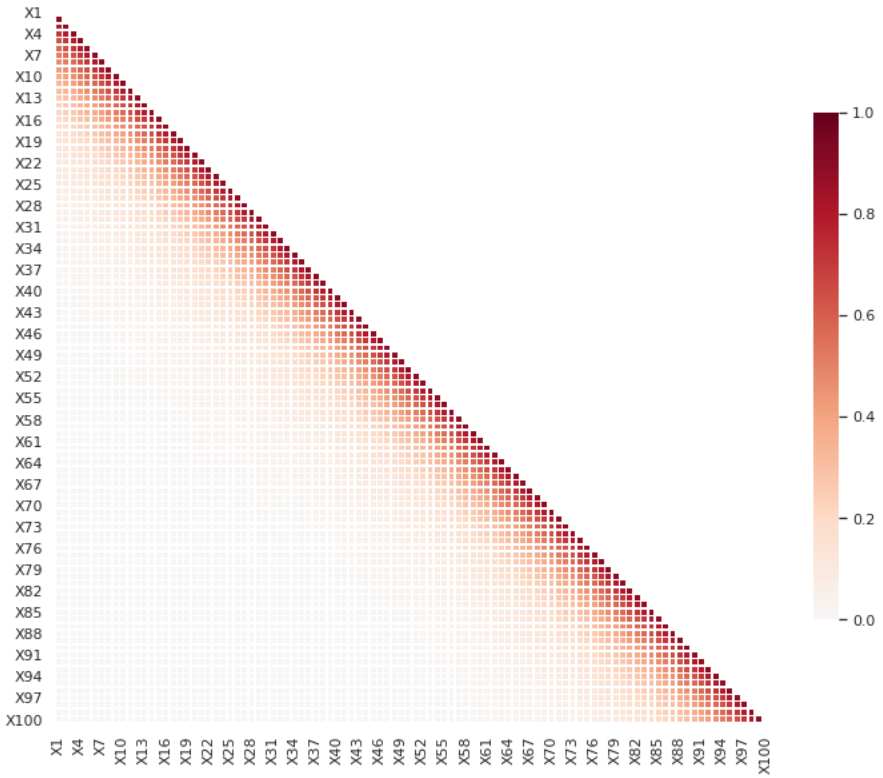
```
df = pd.read_parquet(ROOT / "bld" / "train_simulated.parquet")
X = df.drop("Y", axis=1)
y = df["Y"]
```

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.

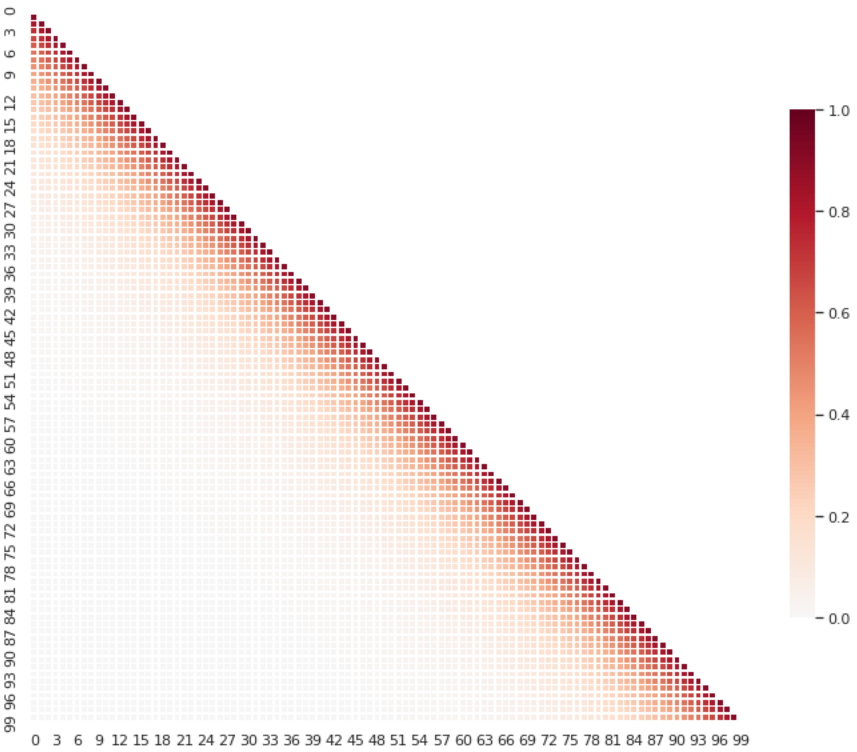
Looking at the correlation between features we notice that the correlation structure looks very similar to the covariance structure of a Matérn covariance function. This can be seen especially well when considering a correlation heatmap of the observed features.

```
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?

But how can we generate *uniform* distributed data with such a correlation structure?

Let Z denote a K (100) 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 a simple algorithm. For this let Σ denote the above Matérn covariance matrix (parameterized with $\ell = 0.1$ and $\nu = 0.5$).

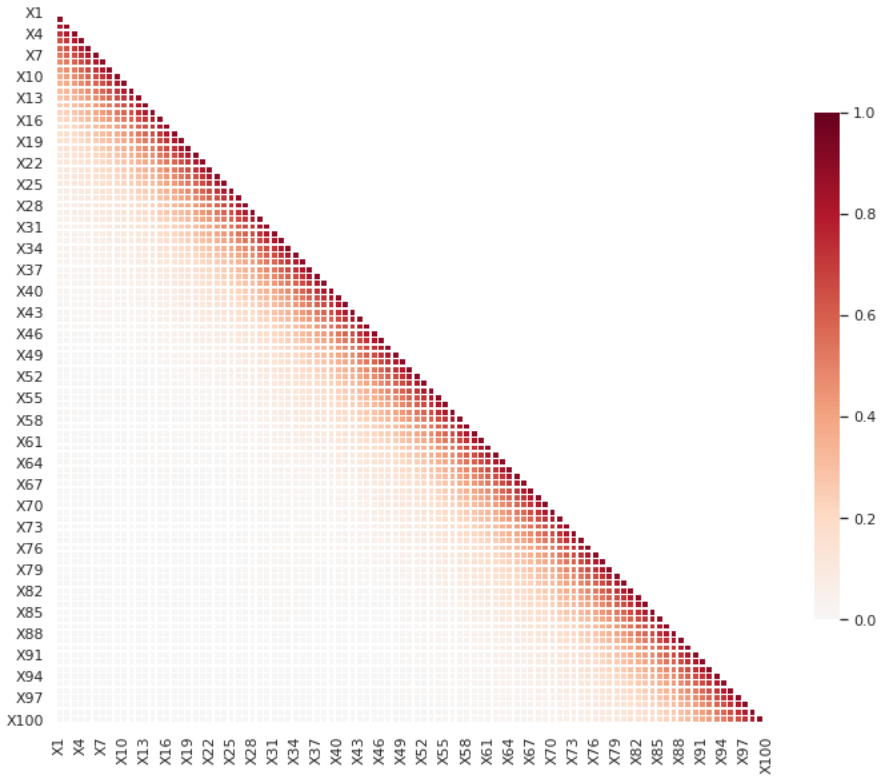
Algorithm:

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

where Φ denotes the standard normal cumulative distribution function.

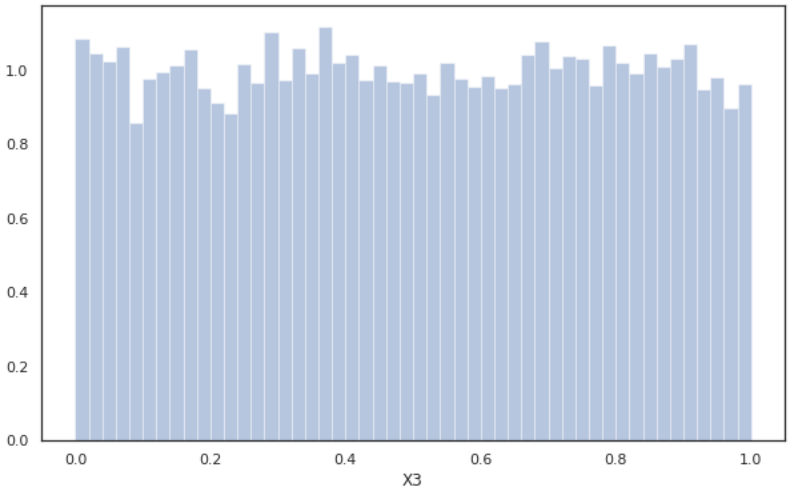
To check wether this works let us simulate a smalle data set in this fashion.

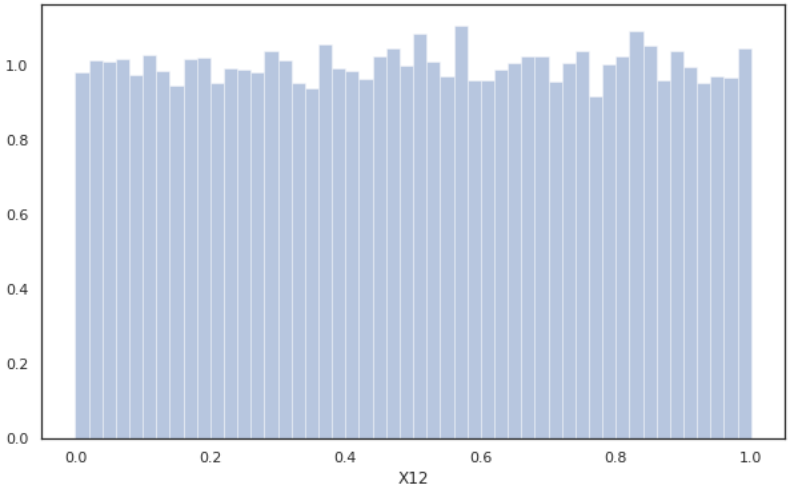
```
n = 25_000
mvnormal = multivariate_normal(cov=Sigma)
Z = mvnormal.rvs(n, random_state=0)

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

shared.correlation_heatmap(XX)
```

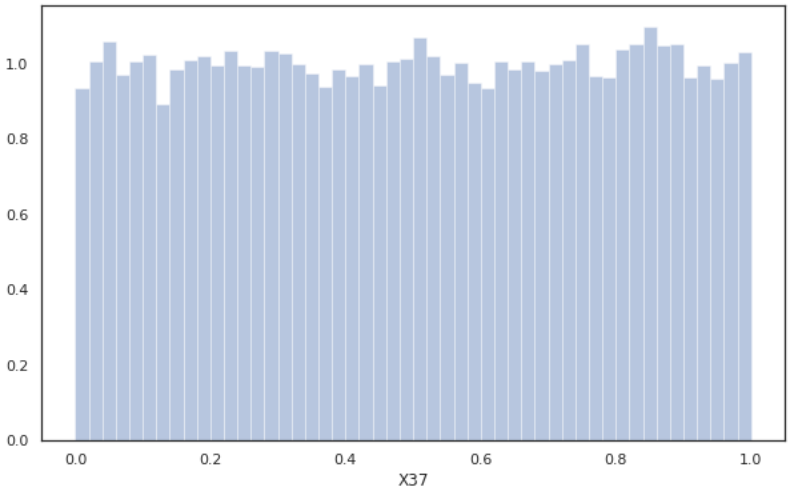


And checking that each column is really uniform distributed by considering a histogram (here we select a few columns randomly since looking at all 100 is too annoying..)

```
for col in ["X3", "X12", "X37"]:
    sns.distplot(
        XX[col],
        bins=50,
        norm_hist=True,
        kde=False,
        hist_kws={"range": [0, 1]}
    )
    plt.show()
```







Recap - Features

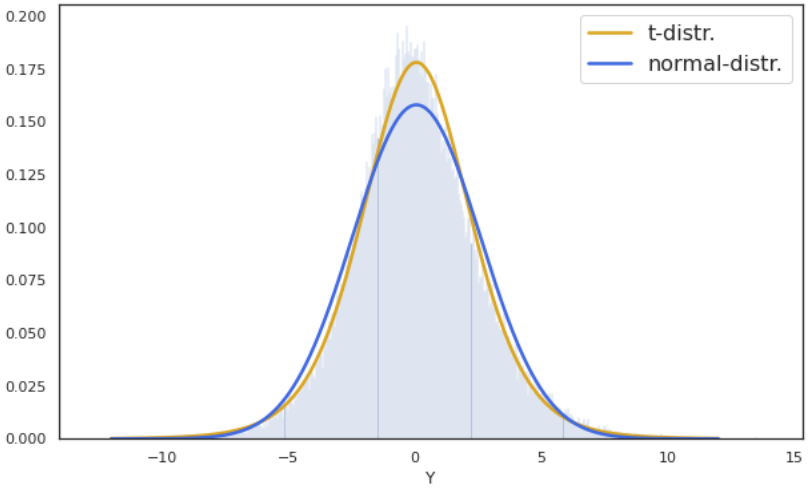
We have seen that the covariance/correlation structure of the features is very similar to a Matern covariance. A major component of feature analysis is to check wether one can reduce the dimensionality of the feature set. As all our features are distributed uniformly and the correlation structure does not admit any grouping we cannot apply classicaly techniques 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 (either via direct or indirect effects).

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.

```
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 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 ($k = 1, \dots, K$) 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")
```

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 strucutre 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 us consider 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 and simulate an additive error term, with which we then compute the outcomes. 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. If this is the case it only remains to find these few relevant features, which would stand out in the sense that the norm of their coefficient values is a local maximum on the discrete space $\{1, \dots, K\}$.

Quantitative Analysis

To get a more quantatitive understanding of which features matter we 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 an third-degree ridge regression for each feature dimension and compare

I wont dive too deep into each approach as in the end I decided to pusue a very different route. For each approach one can imagine fitting an unregularized model on the subset of selected features in a second stage. 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.

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.

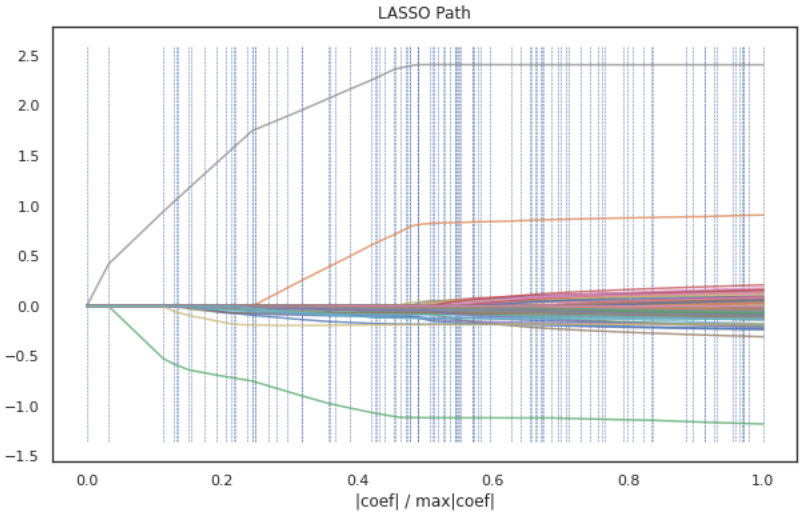
a: first-degree

```
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

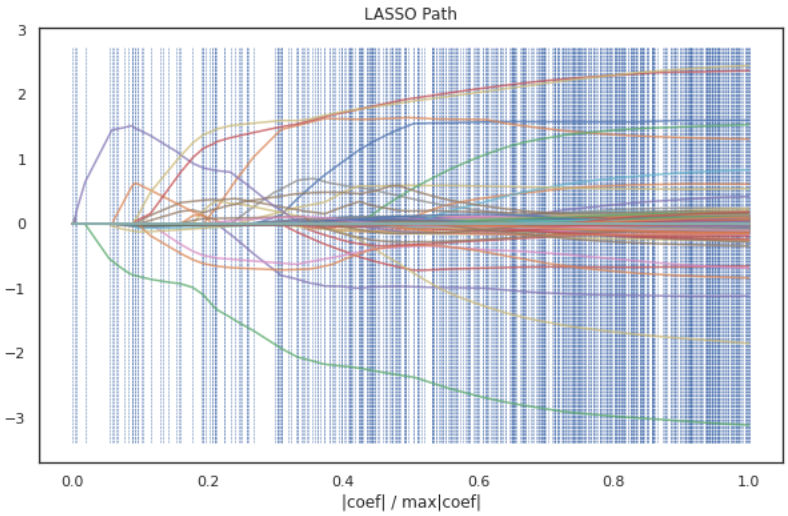
```
poly = PolynomialFeatures(degree=2, include_bias=False)

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']
```

```
_, _, coefs = lars_path(XX.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()
```

```
.
```



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 the features. I use 5-fold CV to compute the *optimal* subset and plot the negative mean squared error against the number of features.

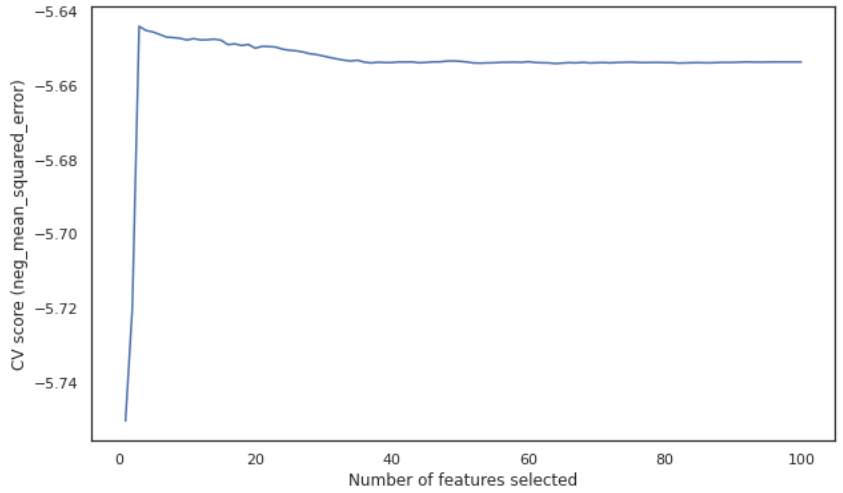
Note.

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, say, 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

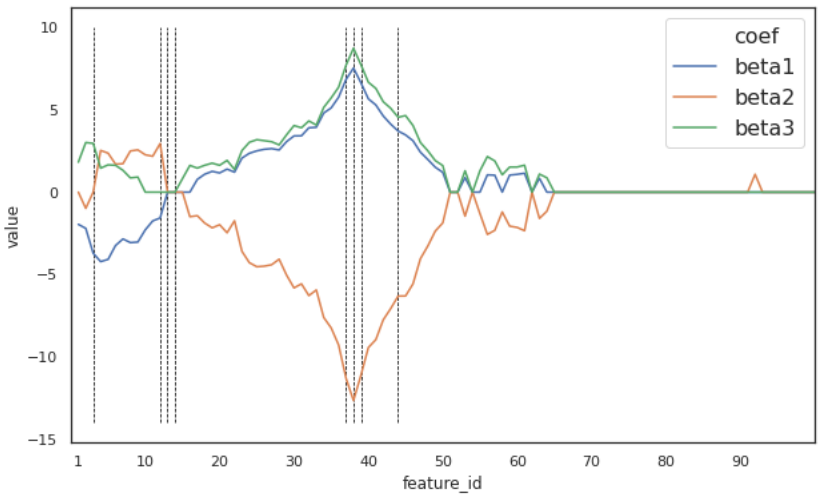
In this last approach the overall goal is to fit a third-degree linear model with l2 regularization, i.e. a ridge regression, for each feature and plot the coefficient values with the feature index on the x-axis. In particular we fit the model $Y \sim \beta_0 + \beta_1 X_k + \beta_2 X_k^2 + \beta_3 X_k^3$ using an l2 regularization for each $k = 1, \dots, K$ and set all coefficients which are below some threshold to zero. The results are compared across k. The regularization parameter is chosen via 5-fold cross validation for each k independently.

```
def get_ridge_coefs(X, y, order=3):
    """Fit a Ridge regression model using cv."""
    XX = PolynomialFeatures(degree=order, include_bias=False).fit_transform(X)
    ridge = RidgeCV(cv=5).fit(XX, y)
    coef = ridge.coef_.flatten()
    return coef
```

```
order = 3
threshold = 0.75

df_coef = pd.DataFrame(
    index=range(1, len(X.columns) + 1),
    columns=["beta1", "beta2", "beta3"]
)
for i, col in enumerate(X):
    coefs = get_ridge_coefs(X[[col]], y, order=order)
    df_coef.loc[i+1, :] = coefs

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.lineplot(
    x="feature_id", y="value", hue="coef", data=df_coef
)
ax.vlines([3, 12, 13, 14, 37, 38, 39, 44], ymin=-14, ymax=10, color="black", lw=0.75,
linestyle="dashed")
plt.legend(prop={'size': 16})
plt.xlim(0, 100)
plt.xticks((1,) + tuple(range(10, 100, 10)))
plt.plot()
```

```
[]
```



The dashed black line 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.

Final

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

In particular 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 the (Python) implementation. Note also that in the very last subsection I include a table 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)
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 which provides an intuitive API for the powerful tensorflow package. 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, 25 hidden nodes for the second layer and 10 hidden nodes for the third to seventh layer. 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 Krizhevsky et al. 2012. As overfitting is a big problem with deep networks I employ a popular technique called dropout

regularization to mitigate this effect, see Srivastava et al. 2014. Dropout leads to neurons in a layer being randomly deactivated for a single epoch during the backpropagation, which in turn leads to neighboring neurons not developing a relationship that is too strongly dependent, which in turn 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 only very few features are relevant. However, I also realized from playing around with some parameters and the validation error of the linear model that some effects must be highly 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 the relevant features. And by using the other additional 5 layers the network can 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 different 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)

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

```
(Neural Network) MSE: 5.325447972161629
```

3. Two-stage Random Forest with Feature Selection

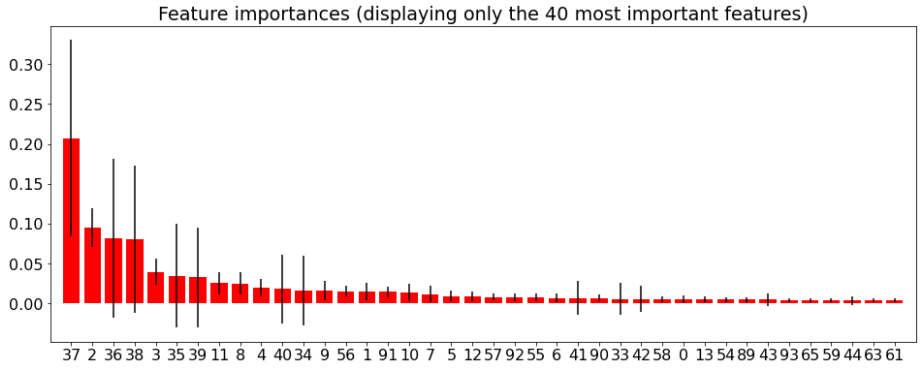
In this two stage procedure I first fit a random forest on the full set of features. I then consider a feature importance measure, which can be automatically calculated from the random forest fit. Using this I select the 30 *most* important features and fit another random forest on this subset of features.

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 am using is a specific variant of a gradient boosted tree. In particular the version implemented in the catboost package, which differs slightly from common implementations. Next I will introduce the concept of gradient boosting with a particular focus on tree weak-learners. Afterwards I show how to fit a model using

`catboost`. Note that my final prediction submissions are not made with this specific model since here I only use my training sample. The final predictions are made with the script final_prediction.py.

Theory

Write about the theory of Catboost HERERERE!!!

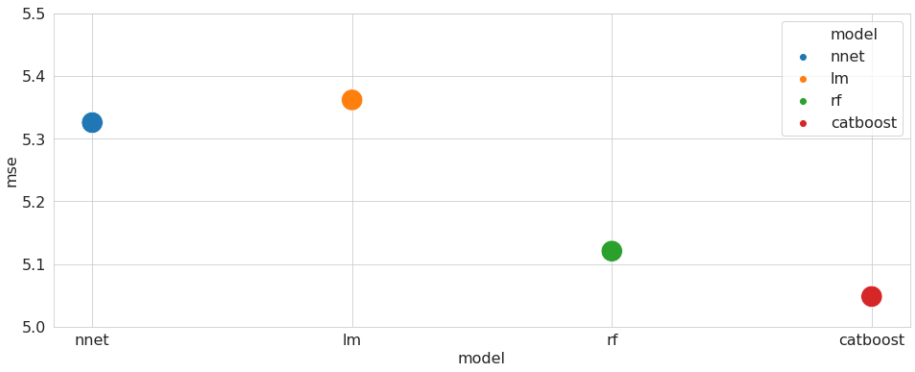
Implementation / Application

```
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 = ["nnet", "lm", "rf", "catboost"]
mses = [mse_nnet, mse_lm, 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)
```



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.

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 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	beme	bem
0	1965-01-31	0.461364	0.513089	0.282723	0.787958	0.319372	0.172775	0.484293	0.60
1	1965-01-31	0.542868	0.240838	0.774869	0.939791	0.340314	0.183246	0.232984	0.30
2	1965-01-31	0.249849	0.633508	0.096859	0.222513	0.884817	0.785340	0.774869	0.78
3	1965-01-31	0.371568	0.439791	0.463351	0.903141	0.434555	0.112565	0.494764	0.64
4	1965-01-31	-0.177803	0.654450	0.335079	0.704188	0.958115	0.848168	0.549738	0.79

The original data set contains 1_629_155 observations of stock returns including 63 features. One of these features 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 given 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 on the data I cleaned it in several ways. First of all I transformed the date column to a year column. I then dropped all observations older than 1990. And at last I dropped all observations which had absolute stock returns greater than 6. 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 smaller sets by year. For each year I split the smaller 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.

Final

My final prediction model is build on the transformed data set from the previous section. That is, I ignore that the time index contains special information and act as if it were only another features. 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

from catboost import CatBoostRegressor

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: 0.9524997479334116
```

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']
```

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: 0.9479932782312956
```

By Tim Mensinger