ML Exercise 2

Regression

Group 8:
Alexander Leitner, 01525882
Mario Hiti, 01327428
Peter Holzner, 01426733

Datasets

- Moneyball
- Superconductivity
- Metro

✖

Regression Models

- K-Nearest Neighbors (KNN)
- Random Forest (RF)
- Simple Gradient descent (SGD)
- Decision Tree (DT)

Dataset	samples	dimensions	Nominal/ordinal	missing values
Moneyball	1232	15	mixed	~20%
Superconductivity	21263	82	numeric	no
Metro	48204	9	mixed	no

Experiment setup:

train-test-split: 70-30 (%)
Cross-Validation via (randomized) Shuffled splits, so each model variation is trained on the exact same training data.
number of splits k: 10

Language + Packages:

Python + numpy, pandas, sklearn (see Pipfile)

Algorithms

Gradiend Descent

Goal:

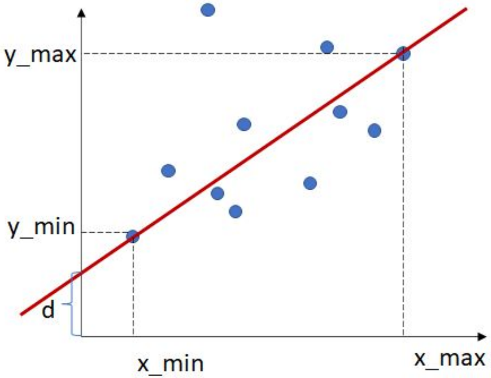
find c, w_i for $y_{pred} = c + x_1w_1 + w_2x_2$ … such that RSS(c, w_i) is minimal

3 Steps:

- Initialize weights
- Iterate using gradient descent
- Making predictions

GD - Step 1: Initialize c, w_i

- for linear regression all c, w_i will eventually converge if α is sufficiently small
- some w_i will converge better as they are already closer to the minimum
- Knowledge of the dataset can help finding better starting values!



- Therefore the following method is used to initialize w_i

For every feature x in the data:

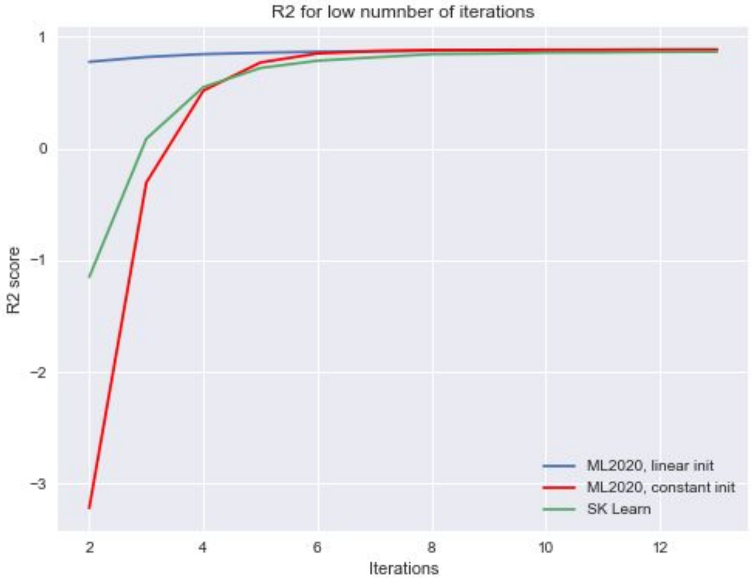
1. find the two "outermost" points x_{min}, x_{max}
2. perform linear interpolation for those two points
3. calculate the slope k and offset d

Use: $w_i = k_i$ and c = average(d)

GD - Step 1: Initialize w_i

Convergence comparison of the R2 score for low numbers of iterations using the money ball dataset for:

- SK SDG Regressor
- Our algorithm using $c = w_i = 0$
- Our algorithm using linear interpolation



GD Step 2 - Iterate

1. make prediction y_{pred} = w X^T + c
2. calculate the residual y - y_{pred}
3. calculate gradients (see next slide)

$$grad_c = \frac{\partial}{\partial c} RSS \qquad grad_{w_i} = \frac{\partial}{\partial w_i} RSS$$

4. calculate new c, w_i

$$c = c - \alpha \cdot grad_c \qquad w_i = w_i - \alpha \cdot grad_{w_i}$$

5. go back to 1) or stop if maximum number of iterations is reached

GD Step 2 - derivatives of RSS

As show on the right the derivatives of RSS can be calculated using a sum and a matrix-vector product

Both can be calculated fairly efficiently using numpy

$$RSS = \frac{1}{N}\Sigma\left(y_i - (c + w_i x_i)\right)^2 \qquad residual = y - y_{pred}$$

$$\frac{\partial}{\partial c}RSS = \frac{2}{N}\sum\left(y_i - (c + w_i x_i)\right) \cdot \frac{\partial}{\partial c}\left(y_i - (c + w_i x_i)\right) =$$

$$= \frac{-2}{N}\sum\left(y_i - (c + w_i x_i)\right) = \frac{-2}{N} sum(residual)$$

$$\frac{\partial}{\partial w}RSS = \frac{2}{N}\sum\left(y_i - (c + w_i x_i)\right) \cdot \frac{\partial}{\partial w_i}\left(y_i - (c + w_i x_i)\right)$$

$$= \frac{-2}{N}\sum\left(y_i - (c + w_i x_i)\right) \cdot x_i = \frac{-2}{N} residual \cdot x$$

GD Step 3 - make predictions

To predict we simply calculate

$y_{pred} = c + w_0 x_0 + w_1 x_1 + w_2 x_2 \dots = w X^T + c$

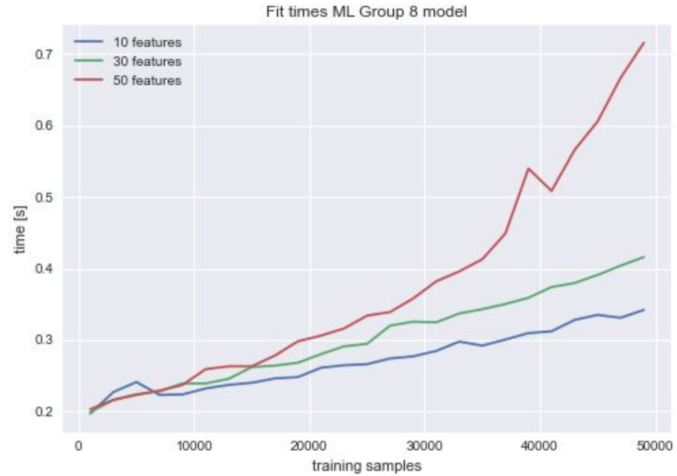
using the c, w_i from the last iteration step

GD Performance

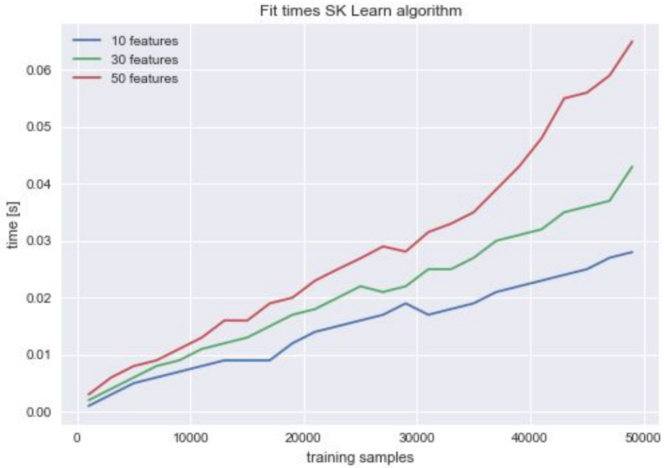
Fit times of our model vs SK SGD Regressor for random data (alpha=0.001, max_iter=1000)

only train times are evaluated because $t_{train} >> t_{pred}$

- SK Learn is ~10x faster
- both have similar time complexity O(N) for low number of features
- Our algorithm gets significantly slower for datasets with large number of samples and many features



Fit times ML Group 8 model



Fit times SK Learn algorithm

GD - lessons learned and improvement ideas

Lessons learned:

- handling matrices can be confusing but checking input dimensions helps
- use numpy whenever you can. replacing sum() with np.sum() has lead to 5x faster performance
- Despite being very simple, linear regression can make really good predictions for some problems

Improvement ideas:

- Stop if there is no notable improvement for several iterations in a row
- use non-constant/adaptive learning rate to speed up convergence if α is too small and avoid diverging residual if α is too big
- use a compiler instead of an interpreter i.e. Cython

KNN - Pseudocode

Initialization/training:

1. Load and store data
2. Initialize model parameters:
    - n_neighbors(=k)… number of neighbors
    - p… order/type of metric
        - i. “0”=maximum norm (usually ∞-norm!),
        - ii. 1=manhattan,
        - iii. 2=euclidean,…
    - dist_func… custom distance function, if wanted
    - algorithm… algorithm used for computing the nearest neighbors (brute force, [kd_tree, ball_tree]*)
3. (If applicable) Prepare stored data for nearest neighbor algorithm
    - for search strategies such as kd_tree, ball_tree (not implemented)

*We did not implement these two, only brute force.

Prediction:

1. Input sample (features)
2. Compute pairwise distances of input to stored data
    - compute difference vector
    - compute distance/norm of difference vector
3. Search for k nearest neighbors = k closest training samples with respect to distance function
4. Retrieve stored regression value for these k samples
5. Compute prediction output as average of k nearest neighbors
    - Common: mean-value (weights=”uniform”)
    - Weighted mean**: usually by distance (weights=”distance”)

**Not implemented, only uniform weighted mean-value

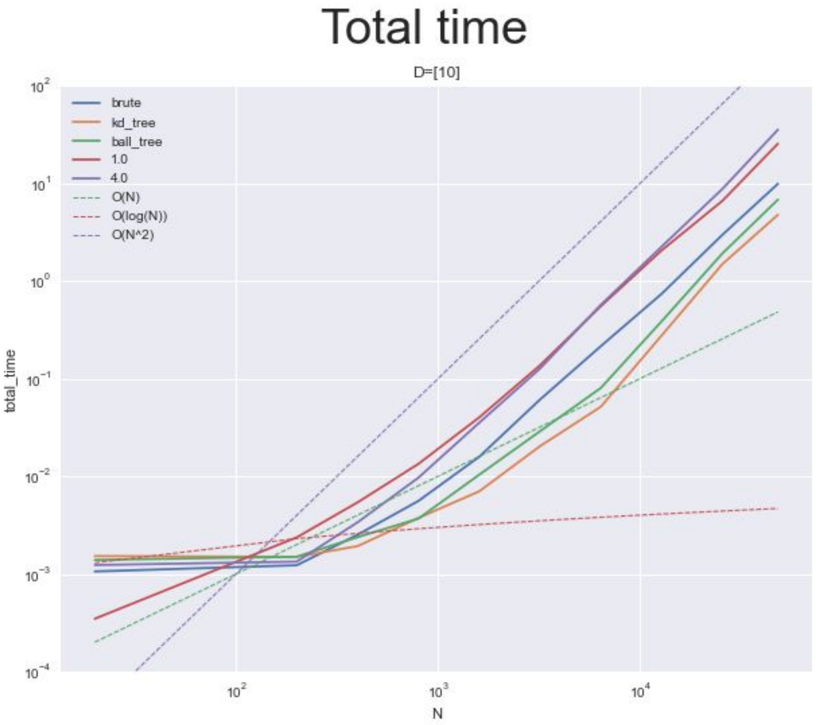
KNN

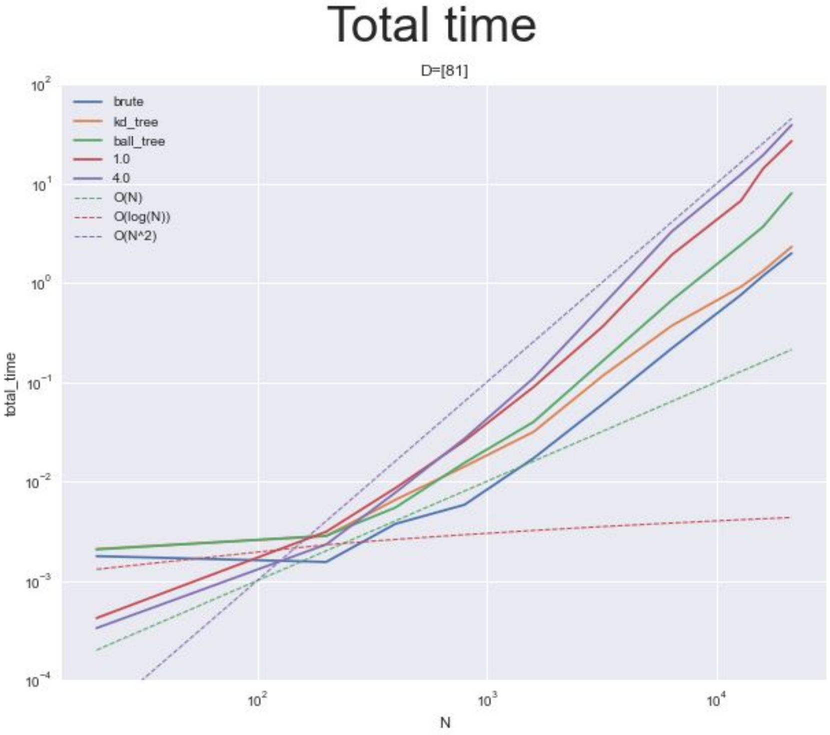
Below, the single-threadedtotal time (=train + inference time) is shown for our and sklearn KNNRegressor. We extracted subsets of size N of 2 datasets (left Metro, right Superconductivity) to benchmark the model efficiency/complexity.

Parameters: k=5, norm=euclidean (2-norm)

sklearn-KNN: algorithm = [brute, kd_tree, ball_tree]

myKNN (our implementation): chunk_size=[1, 4]

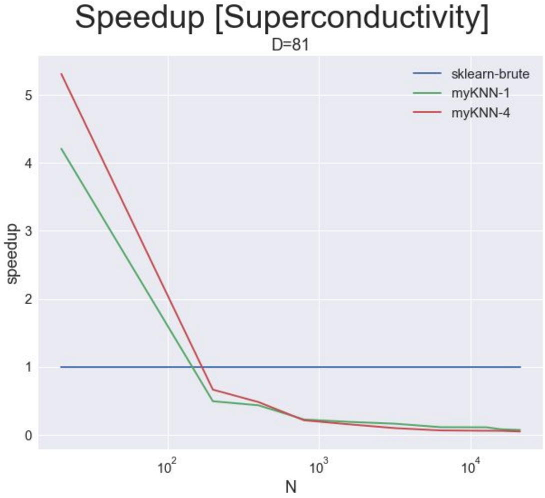


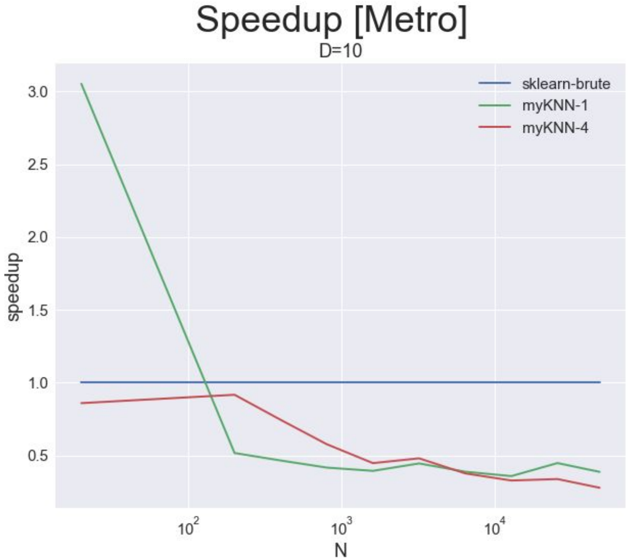


KNN - Efficiency

Speedup plots are in reference to the sklearn implementation using the brute-force algorithm.
Not only the dataset size N, but also the number of features D impacts the efficiency/performance.
Higher D seems to increase our speedup for very small N<1e2, but also results in even worse slowdown for large N>>1e2.

Overall, our KNN implementation performs very well for small datasets (faster than sklearn) and is even useable for larger datasets (~30s total time), such as the Metro (N≈40k, D=10) and Superconductivity (N≈20k, D=81) sets.
The model performance/quality is otherwise identical to sklearns, see later slides.





KNN - Notes on implementation

numpy functions used for (vectorized) numerical operations*:

1. distance: np.linalg.norm**("p>1"-norms) and np.max (inf-norm)
2. search k-smallest: np.argpartition** (retrieve indices of k nearest training samples) + np.take_along_axis for retrieving
3. average: np.mean

*Rule no. 1 of numerical python: Don't try to beat numpy!
**Sidenote: We also tried out scipy.spatial.distance.cdist to compute pairwise distances out of curiosity (it seems that sklearn also uses a similar implementation for their brute force search), but the inference time using that was worse by a factor of 5 then our current version.

Notes:

- Computing k-nearest neighbors per prediction sample was actually fastest (pairwise)***
- Algorithm profile:
  >99% of time spent on difference vector + distance computation (for large datasets)
  rest is almost negligible
- Distance function:
  - major impact on inference time:
    e.g. time(manhattan) / time(euclidean) = 2 / 3
  - impact on model performance:
    mostly dependent on data
- Overall: we achieve the same model performance with 10x the runtime as sklearn's KNNRegressor

***sklearn does something similar

KNN - Issues

Training: no issues as we did not do any preparation as needed for kd_tree and ball_tree

Predicting + training on large datasets:

N…total number of samples (Assume: N=n+m)
n…number of samples in training set
m… number of samples in test set
D…number of features

To predict on m test samples, m*n distances (=norm of vectors) need to be calculated based on vectors with D elements.
Hold all difference vectors at same time: n x m x D 3d-array

- example: 40k (Metro) samples total
  n=30k, m=10k, D=10, sizeof(double)=8byte
  → memory req = n*m*D*sizeof = 24GB
- PCs are often limited to <=16GB of RAM
  → swap used on SSD/HDD → performance terrible

Solution for large datasets: ***"chunking"***

- compute k-nearest neighbors in chunks/subsets of the m test samples (set by attribute chunk_size)
- computing the neighbors per test sample was actually fastest (chunk_size=1)

Superconductivity

Description/Preprocessing

Description:

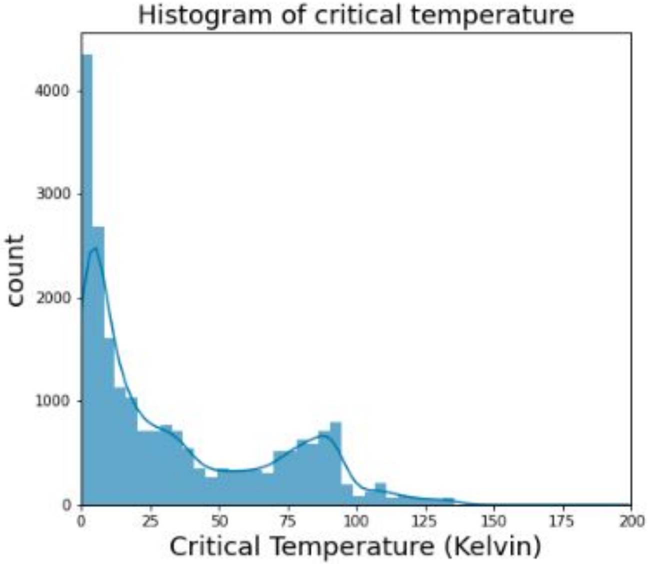
number of features D: 81 (all numerical)

number of samples N: 21k (no missing values)

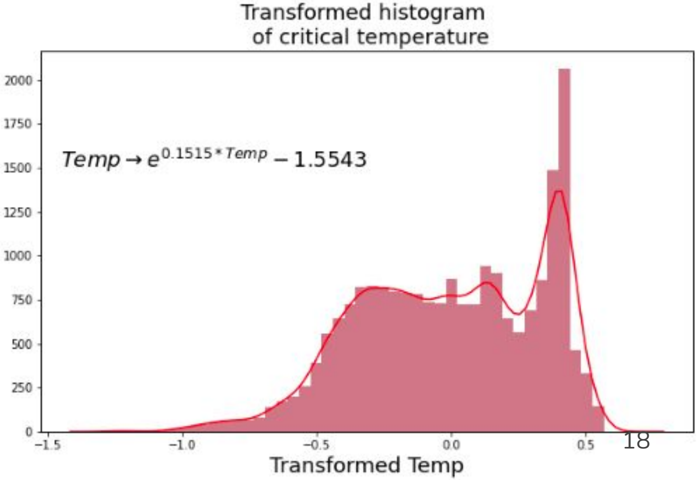
Contains information about various molecules, such as:

atom mass, Valence, etc.

critical temperature (target)



The original target (critical temperature) distribution is shown on the top right. The distribution is unfortunately not well suited for (all) regression models. We therefore tried to transform the target to look similar to a normal distribution. The transformed target distribution and the transformation function we used is shown in the bottom left.



The features had differing ranges or were of different orders of magnitude. We used a simple StandardScaler to correct this.

No feature selection or other data preparation was done.

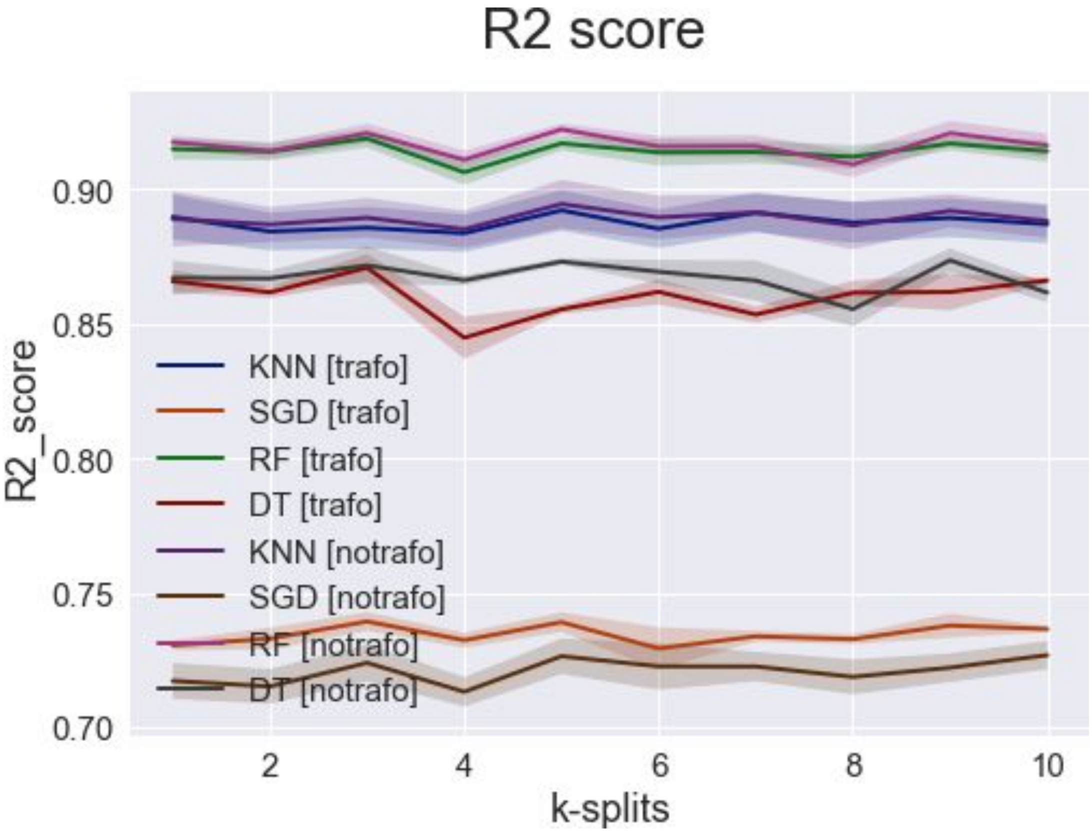
Analysis Transformed vs Original Target

The performance of all models was never negatively impacted by the transformation and only improved the performance of the SGD-Regressor slightly.

We therefore continued to use the transformed data for the rest of our experiments.

In a real world application, one might consider not doing this due to the extra step. The reverse transformation would need to be applied to the prediction output of the model.

Note on (all) graphs in this section:
- each line is the mean of all model variations.
- colored area around each line represents the std-deviation across model variations (hyperparameters)



Analysis SGD

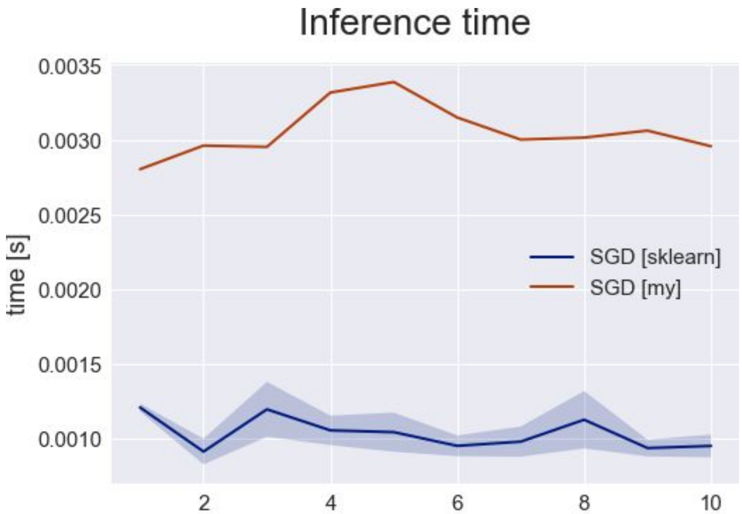
Hyperparameters:
"alpha" : [0.0001, 0.005],
my: "max_iter": [1000],

Overall, the models of the SGD-type seem to not fit too well to this dataset, though the results (R2 score) are not bad.

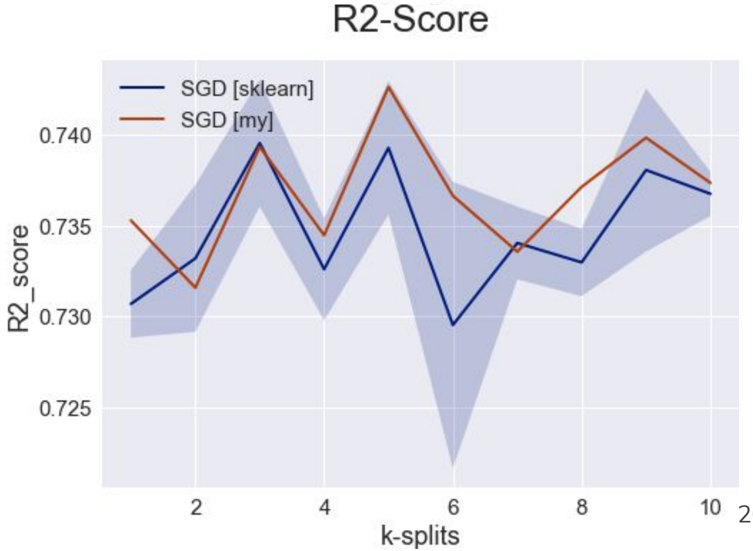
Our implementation results in a similar model or one of similar quality as sklearn’s implementation - the R2-score for both implementations are close to identical.

The lacking optimization of our implementation compared to sklearn’s is of course apparant when looking at the inference and train times (see later slides) needed.

Our implementation is roughly 10x slower than sklearn in training and 3x slower at running predictions - at least for such a large dataset with 21k samples (test set: 30% ≊ 6.3k samples)



Inference time



R2-Score

Analysis KNN

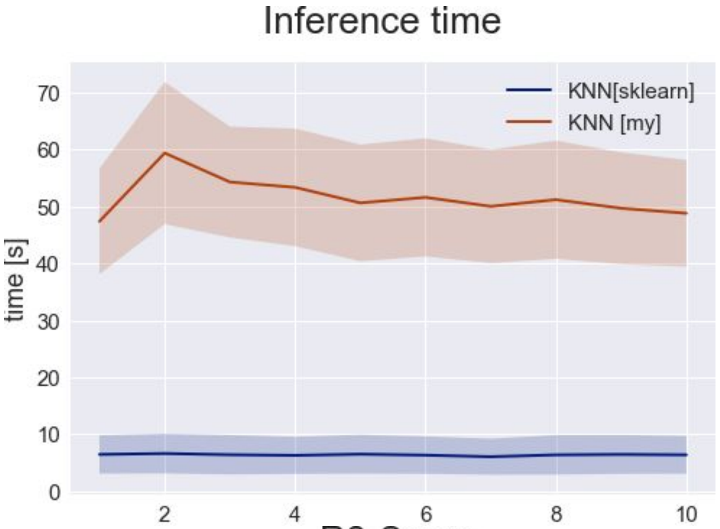
Hyperparameters:
"n_neighbors" : [5, 10],
"p": [1, 2], "weights" : ["uniform"],
"algorithm": ["brute","kd_tree","ball_tree"]

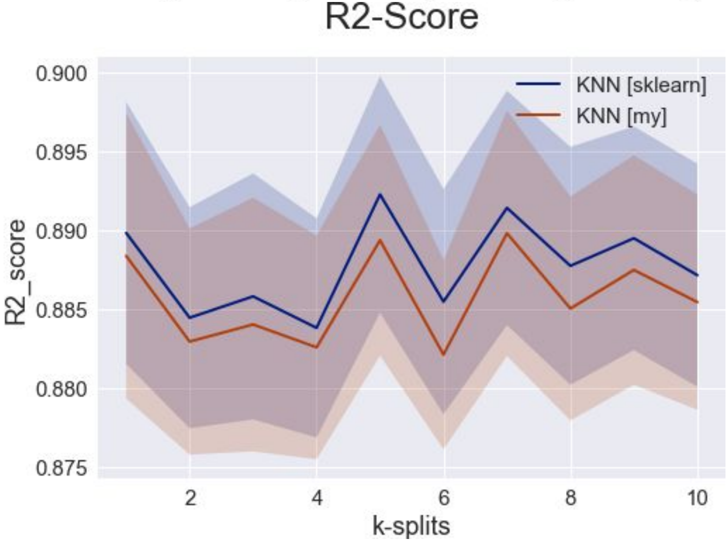
KNN reaches good R2-scores overall.

Our implementation results in a similar model or one of very similar quality as sklearn’s implementation. There is a very minor difference in R2-scores across the board.

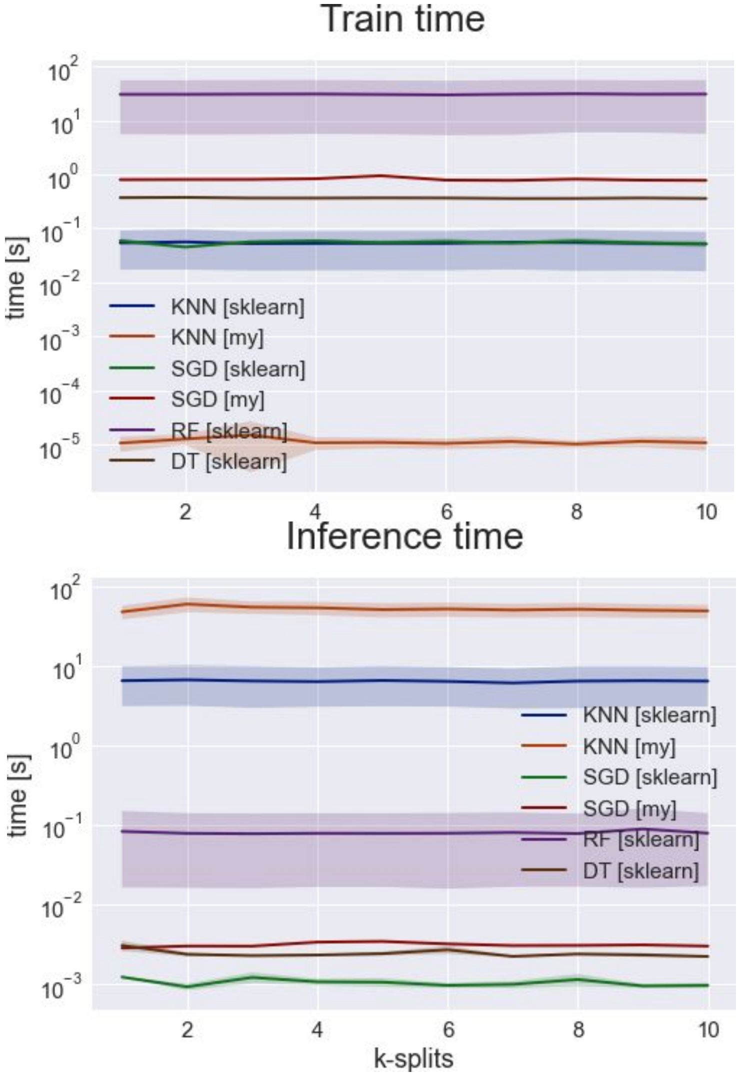
The lacking optimization of our implementation compared to sklearn’s is of course apparant when looking at the inference and train times (see later slides).

Our implementation is roughly 10x slower at running predictions for this large dataset with 21k samples (test set: 30% ≊ 6.3k samples). Our KNN trains multiple magnitudes faster - which is unfortunately irrelevant most of the time.





Runtimes



Left: recorded train and prediction/inference times with a train-test-split of 70-30%, all models used a single-threaded execution mode
The colored area around each line shows the standard deviation across all tested hyperparameter sets.

The recorded train and inference runtimes show the expected results:

- KNN: our implementation trains faster but is slower at predicting than the sklearn-KNN. KNN is overall the slowest model for this (large) dataset
- SGD: our implementaiton is not as efficient as sklearn. Overall SGD is one of the faster models for this dataset.
- RF: The slowest learner, but the prediction efficiency is good (middle of the pack)
- DT: The second slowest learner and second fastest predictor. Training times had very high variance (not plotted, as it would have filled the whole plot)

Runtimes recorded on:
Intel i5 4x1.4GHz (<3.8GHz boost), 16GB RAM, MacOSX

Resumé

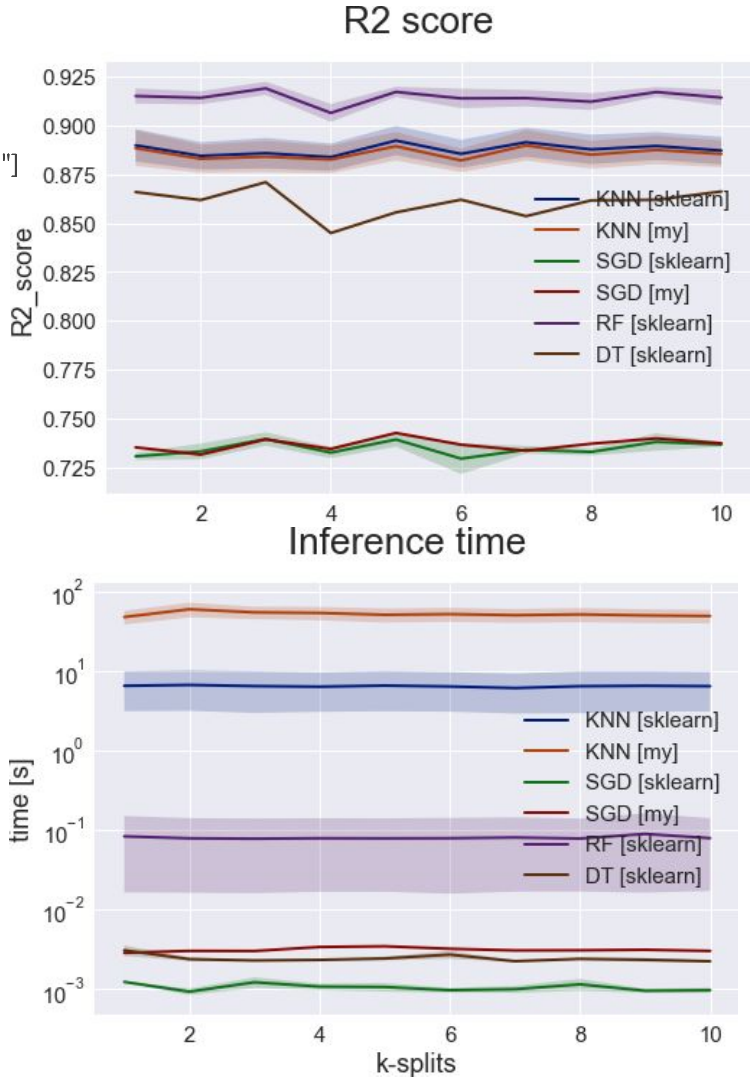
Hyperparameters:
RF: "n_estimators" : [10, 100]
DT: "criterion": ["mse"],
"max_features": ["auto", "sqrt","log2"]

Overall, the RandomForest seems to perform best, consistently reaching a R2-score > 0.9 (top right). It’s also a rather fast predicter (bottom right).

We suspect, that KNN could perform similarly well in terms of prediction accuracy (R2-score), if one searches for the optimal hyperparameters - KNN is close behind the RF regressor.
The complexity of the prediction (runtime) is the real deal breaker compared to RF.

SGD and DT are the clear “losers” as they simply do not reach R2-scores that come closer to the above.

The hyperparameters used for RF and DT are mentioned above.



Metro

Description

Description:

holiday (nominal)

temp (numeric)

rain_1h (numeric)

snow_1h (numeric)

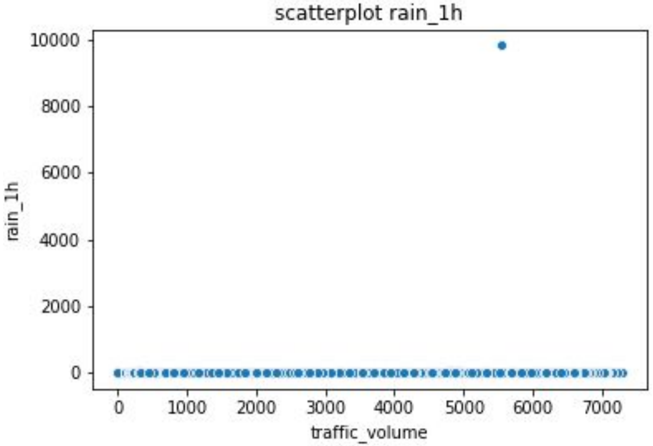
clouds_all (nominal)

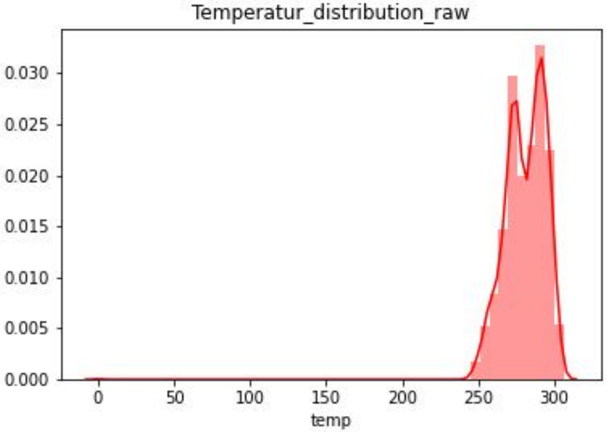
weather_main (nominal)

weather_description

date_time (numeric)

traffic_volume (target)





for the temperatur(K) distribution it looks like there are some out-layers. In some rows the temperature is 0K and that is not possible -> maybe an error at the measuring.

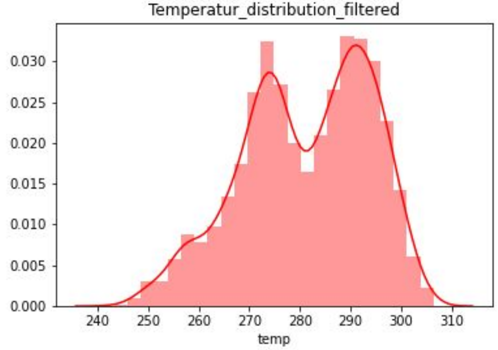
The mean value for the rain_1h is 0.345 and its maximal value is 9439.45. -> outlayer

Preparation

The date column is also rewritten in a format that the model can use -> year, mounth, day and hour (different columns).

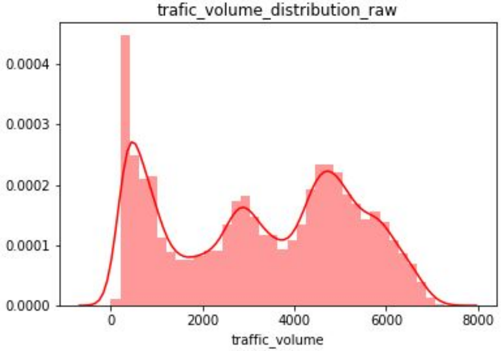
The weather_main and weather_description is coded with numbers.

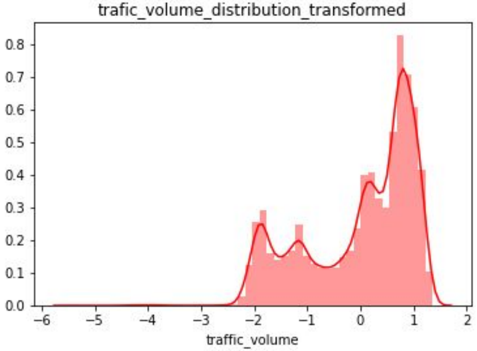
Finally the dataset is splitted in the features (scaled with a standard scaler) and the target.



Simply replace the values with 0K with the mean temperature. The distribution looks fine now.

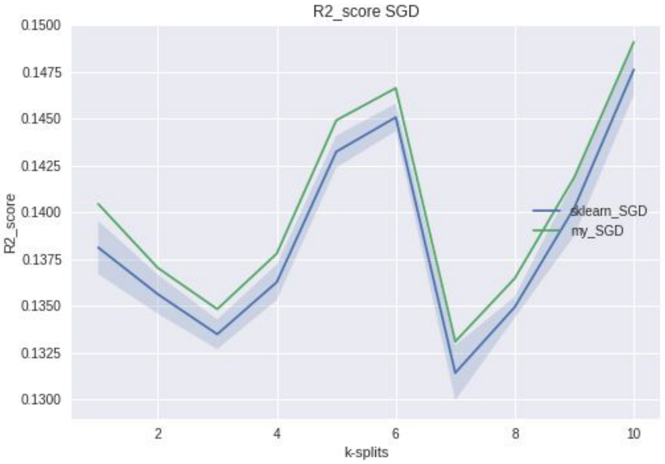
The out-layer in the rain_1h column is dropped.

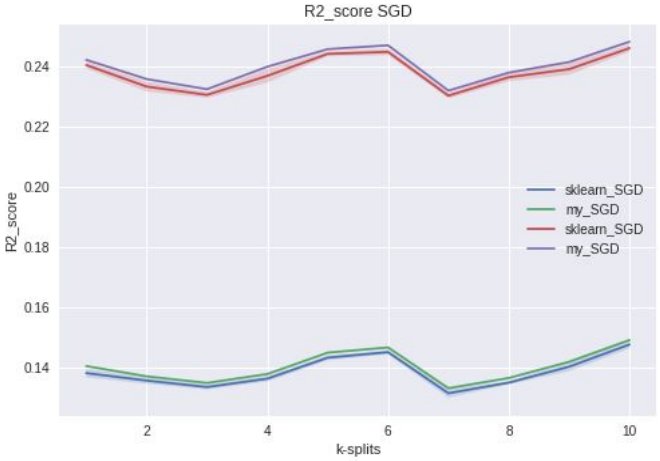




We tried to transform the target into a better shape for the regression models to get better results. We used the same transformation as for the superconductivity target. Especially the linear regressors are sensitive to the "shape" of the target data (the distribution), e.g. SGD. This could affect the score and also the performance of the different models.

Analysis SGD





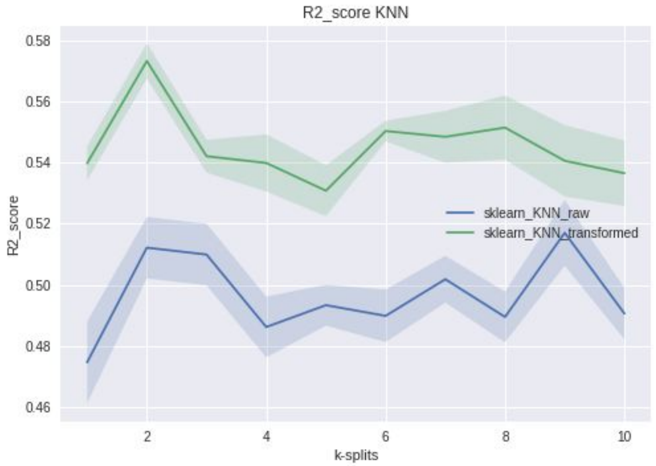
Note on all graphs in this section:
Each line shows the mean value of all model variations across all 10 splits. The area around each line is calculated by adding and subtracting the standard deviation from the mean value. For our implementation the area (std-deviation) is 0.

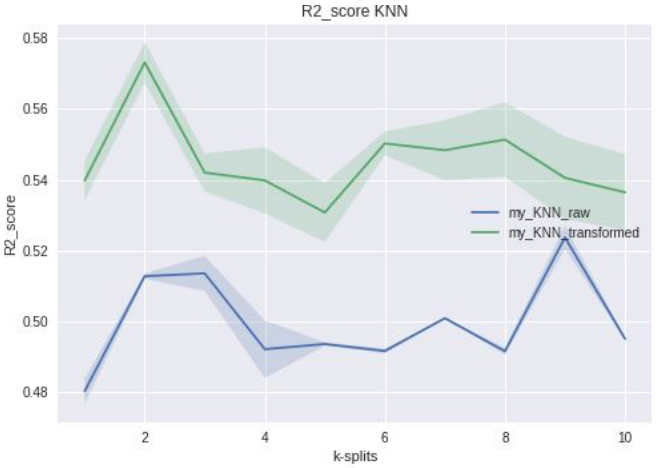
Results:

Hyperparametes:
alpha = 0.0001, 0.00001
max_iter = 1000, 2000, 3000, 5000

Our SGD's score is slightly higher than the Sklearn implementation. The upper graph shows the scores useing the raw (untransformed) target - they are considerably worse. The red and violet line from the lower graph is ~ 10% higher. The RMSE score is for the transformed one always lower 1 and for the raw one 10^6 where the range is from a few hundred to 6000.

Analysis KNN





Results:

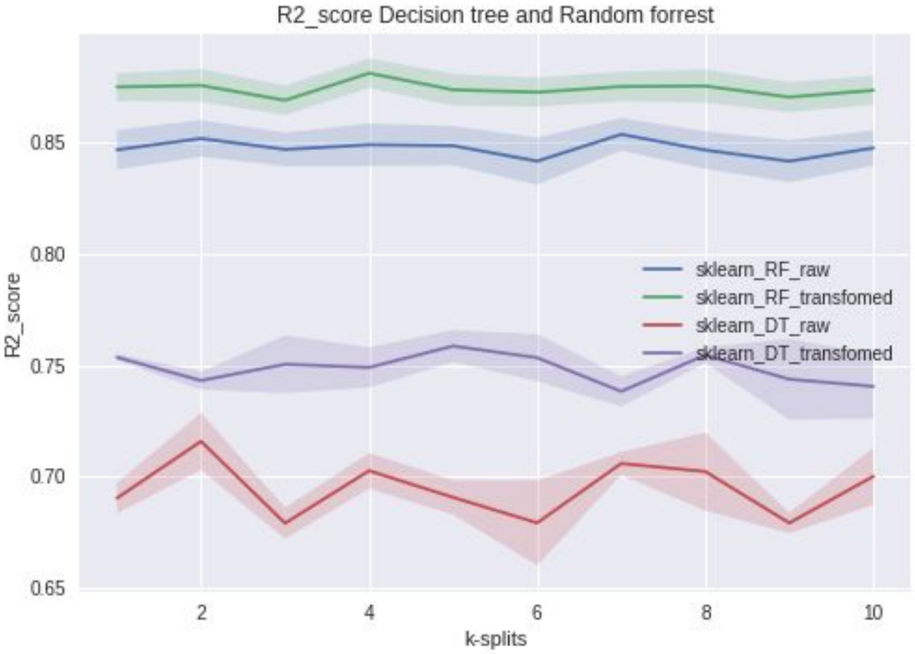
Hyperparametes:
weights = uniform
n_neighbors = 5, 10
p = 2, 3

The score of our implementation and Sklearn's KNN is almost the same with only very minor differences. With the transformed target distribution, we get better results for both KNN implementaitons by ~6%.

The distance function (or rather the order of the metric given by p) influences both the R2-score and the prediction runtime (see later slides), where higher orders of p are usually slower.

The target transformation does help the models here.

Analysis RF + DT



Results Random Forrest:

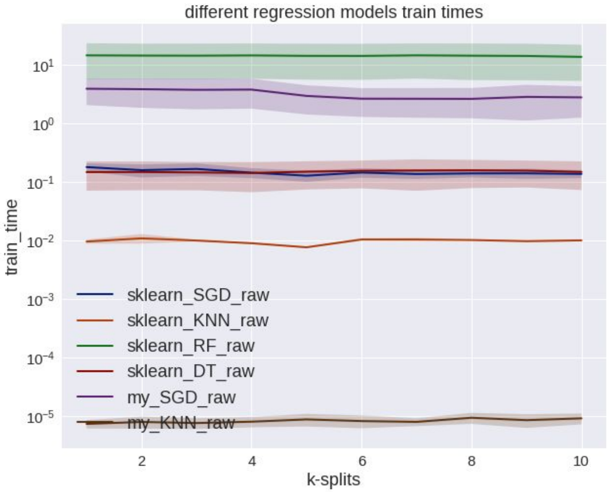
Hyperparametes:
n_estimators = 100, 200
max_features = auto, sqrt, log2

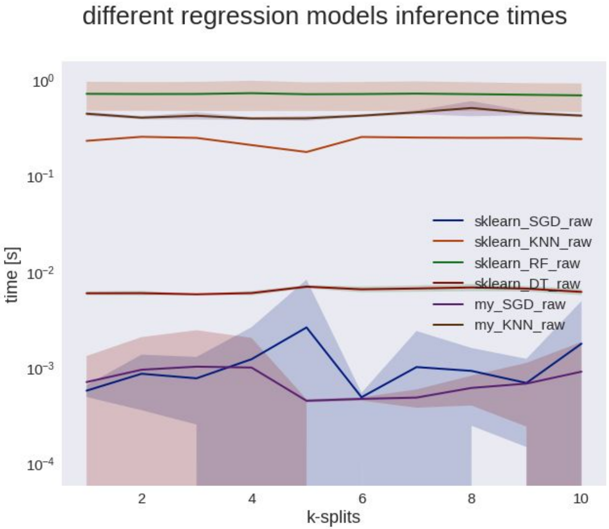
This model achieved the highest score from the othermodels and it has the same offset between the raw andthe transformed target data. The standard deviation in a split is nearly 1%. This could be improved by choosing the n_estimators to a higher number.

Results Decision Tree:

Hyperparametes:
criterion = mse
max_features = auto, sqrt, log2

Again the transformed target achieved a better score with this model and the standard deviation is larger in comparison to the other models. So to choose different parameters is more crucial for the Decision Tree model. Also the score oscillate a lot more than the other models.





Train Inference time

train time:

The standard deviation between the Sklearn SGD and our implementation of the SGD is approximately the same. The Sklearn model is faster ~10 times faster. Our implementation of KNN is approximately 1000 times faster at training than sklearn.

inference time:

In comparison to the traintime our KNN implementation is slower than the Sklearn KNN. The inference time for the Sklearn SGD and our implementation is approximately the same. The Sklearn KNN hasthe biggest standard deviation -> our implementation's (with the same hyperparameters) is not as large. The difference in the standard deviation of the prediction time between Sklearn's (0.25s, too large -> not plotted) and our KNN(~6e-5s) is larger. Sklearn's high deviation here is probably caused by the different metrics/distance functions they use for the nearest neighbor search.

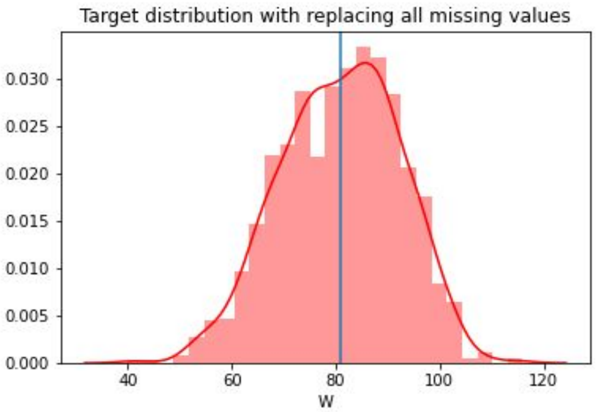
Moneyball

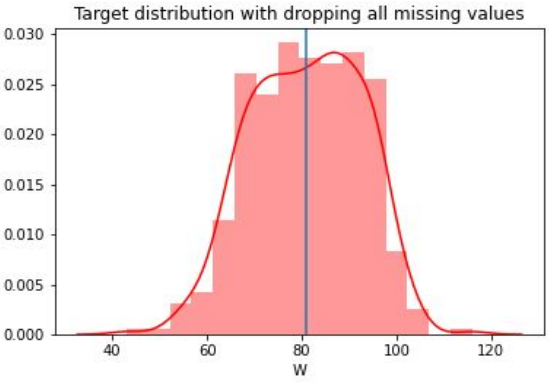
Description/Preprocessing

Description:

- smallest dataset (1,2k samples, 15 dimensions)
- numerical and nominal columns

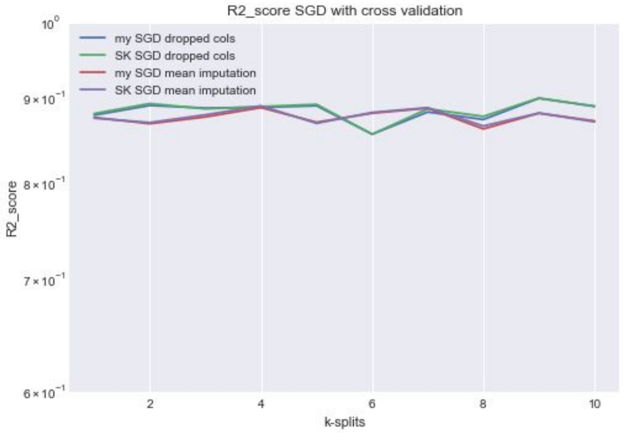
- 2 strategies for preprocessing:
  - drop missing values
  - mean imputation
- The StandardScaler from SK Learn is used for scaling

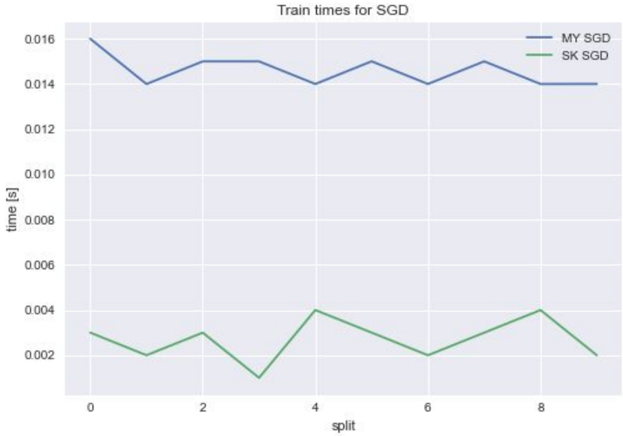




Analysis SGD

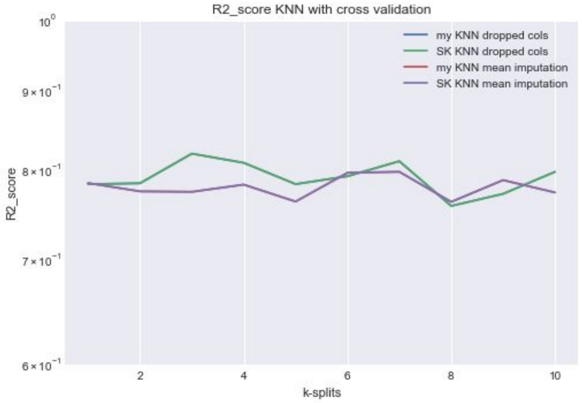
- Overall good performance (R2 score ~0.9)
- mean imputation leads to better results than dropping columns with no data
- R2 score of SK Learn and our implementation are almost is almost identical
- SK learn is ~10x faster

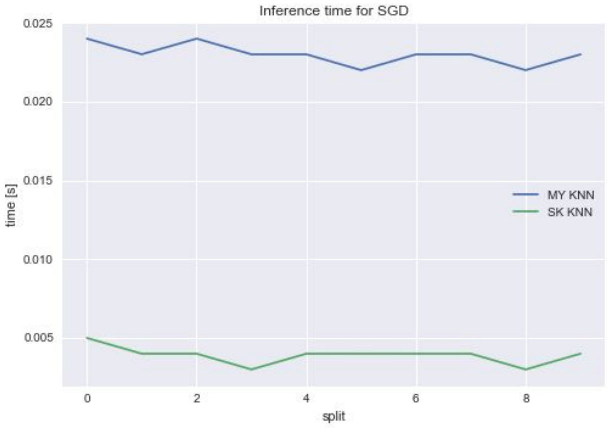




Analysis KNN

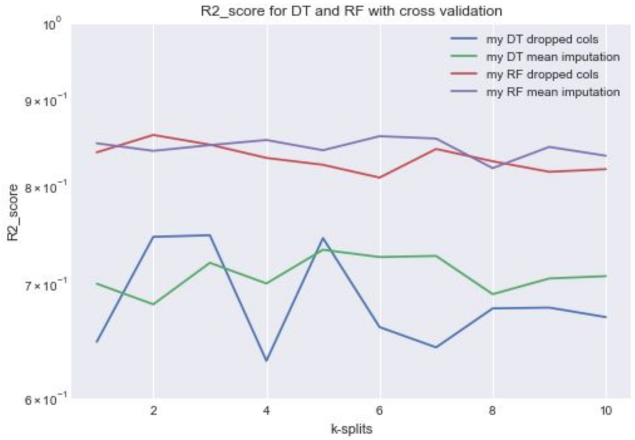
- Predictions of SK learn and our implementation is identical
- R2 score is slightly lower compared to SGD
- SK Learn is ~5x faster

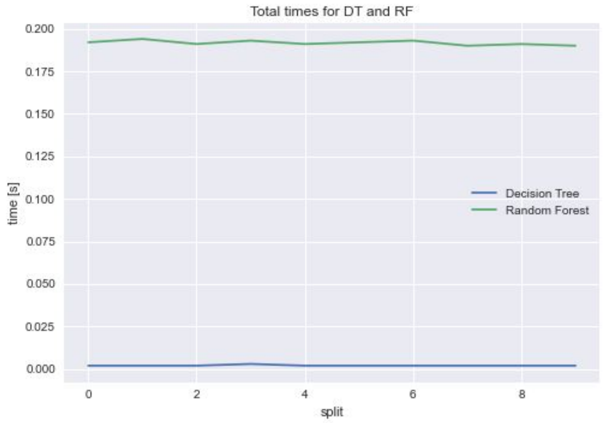




Analysis RF + DT

- Random forest leads to overall better results
- Decision tree is the fastest compared to all other algorithms

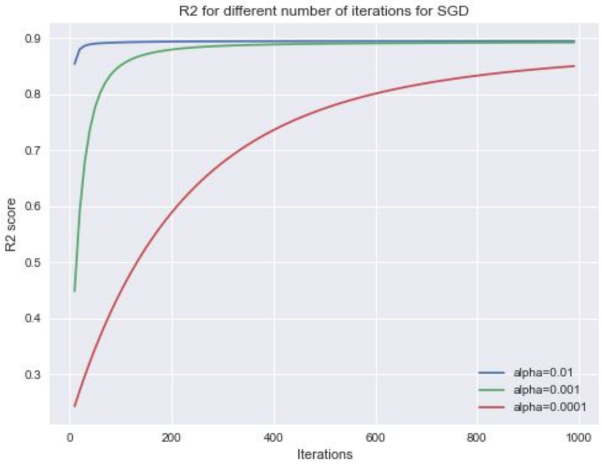


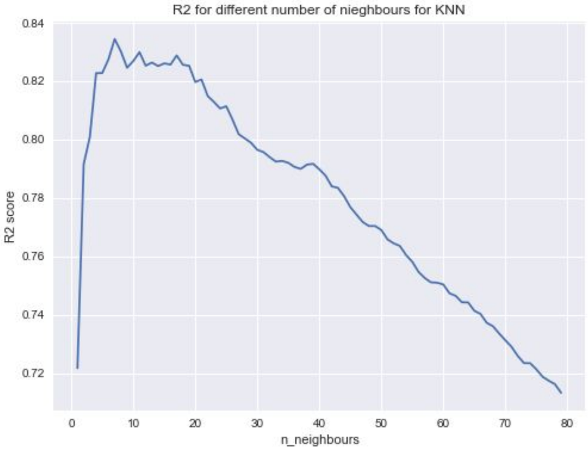


Parameters for KNN and SGD

- SGD will always converge if α is sufficiently small
- Better values for α lead to faster convergence
- if α is too big result will diverge

- 8-20 neighbours lead to best results for KNN





Summary and conclusions

- Overall good results for all models (R2 = 0.7 - 0.9)
- SGD achieved highest R2 scores overall
- mean imputation leads to better results
- SK learn is generally faster than our implementation
- decision tree is the fastest algorithm but due to the size of the data set train times and inference times do not really matter