# DESIGN AUTOMATION CONFERENCE

FROM CHIPS TO SYSTEMS — LEARN TODAY, CREATE TOMORROW

DEC 5 - 9, 2021 ◆ San Francisco, California

# Hands-on ML: Post-layout Capacitance Estimation

Siddhartha Joshi and Brett Shook

With

Sunder Kankipati, Prateek Bhansali, Chandramouli Kashyap

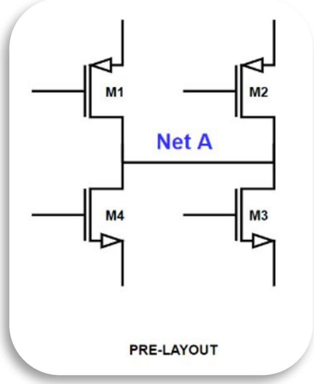
Intel Corporation

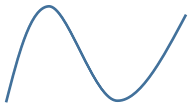
# Tutorial Outline

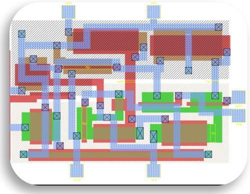
- Introduction:
  - Custom circuit design flow and parasitic estimation
- Machine Learning(ML) Pipeline for Parasitic Estimation
  - Data collection, preparation, feature engineering, training, etc.
- *Accompanying paper: B. Shook, P. Bhansali, C. Kashyap, C. Amin and S. Joshi, "MLParest: Machine Learning based Parasitic Estimation for Custom Circuit Design," 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1-6, doi: 10.1109/DAC18072.2020.9218495.*

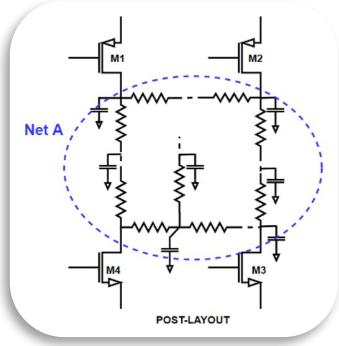
# Custom Circuit Design Flow

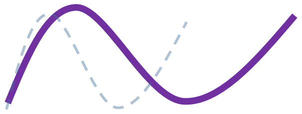
Schematic Design/Migration/ Cleanup → Pre-layout Design and Simulation → Layout and Extraction → Post-Layout Simulation



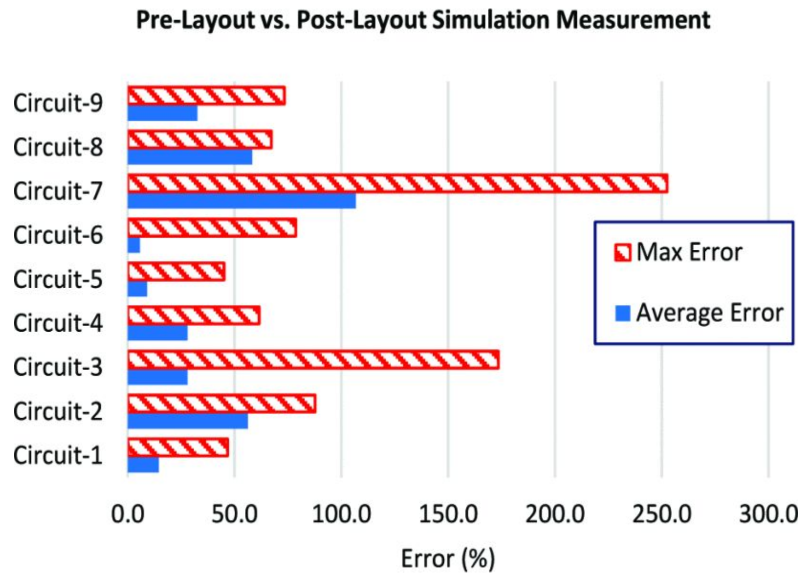




# Problem Definition

- Post-layout design metrics impacted by interconnect parasitic and device layout effects
- Pre-layout vs. post-layout simulation results differ up to 250%*
- Post-lay/pre-lay iterations expensive
  - Causes delay in product schedule



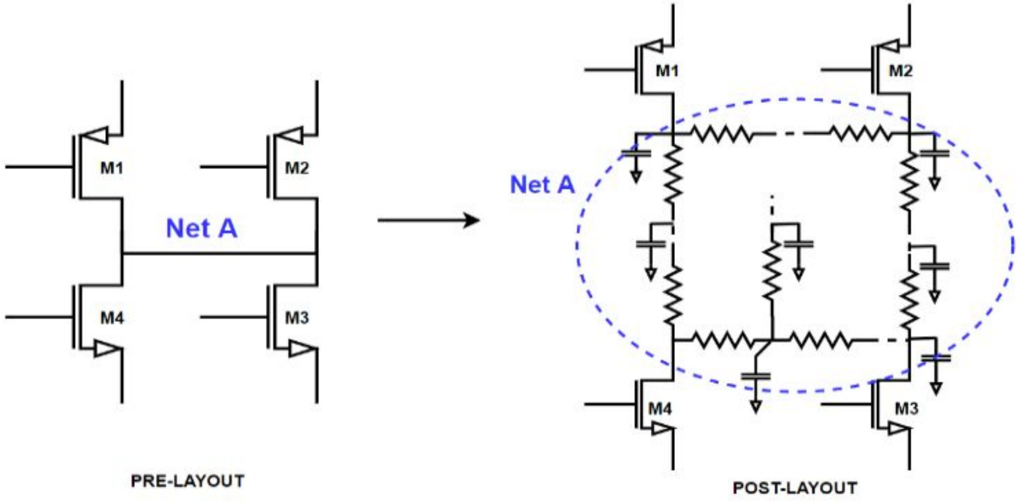

Schematic vs Post-Layout Simulations of Analog Circuits in Intel 10nm Technology [MLParest, DAC 2020]

# Parasitic Estimation

- Designers routinely guess and put explicit resistor and capacitors to model interconnect effects
  - Manual and require a lot of experience
  - Need to maintain separate schematic
- ALS is the holy grail and an active area of research
- Our solution: Automatically estimate parasitics in pre-layout phase and avoid iterations
  - Can we use machine learning? Yes.

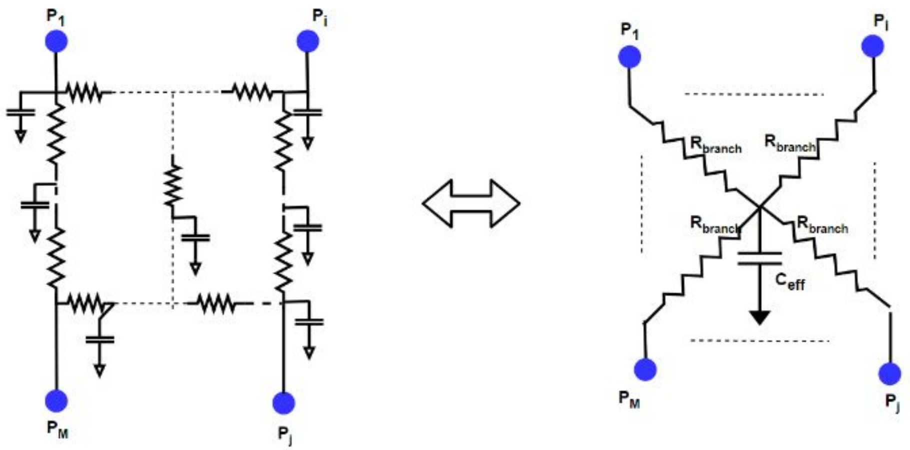
# MLParest: Machine Learning Based Parasitic Estimation

- Given a pre-layout schematic
  - Leverage existing data and estimate interconnect parasitics
  - Interconnect parasitics should be usable in standard circuit simulation flow
  - Should not increase SPICE runtime



# MLParest Design/Modeling Choices

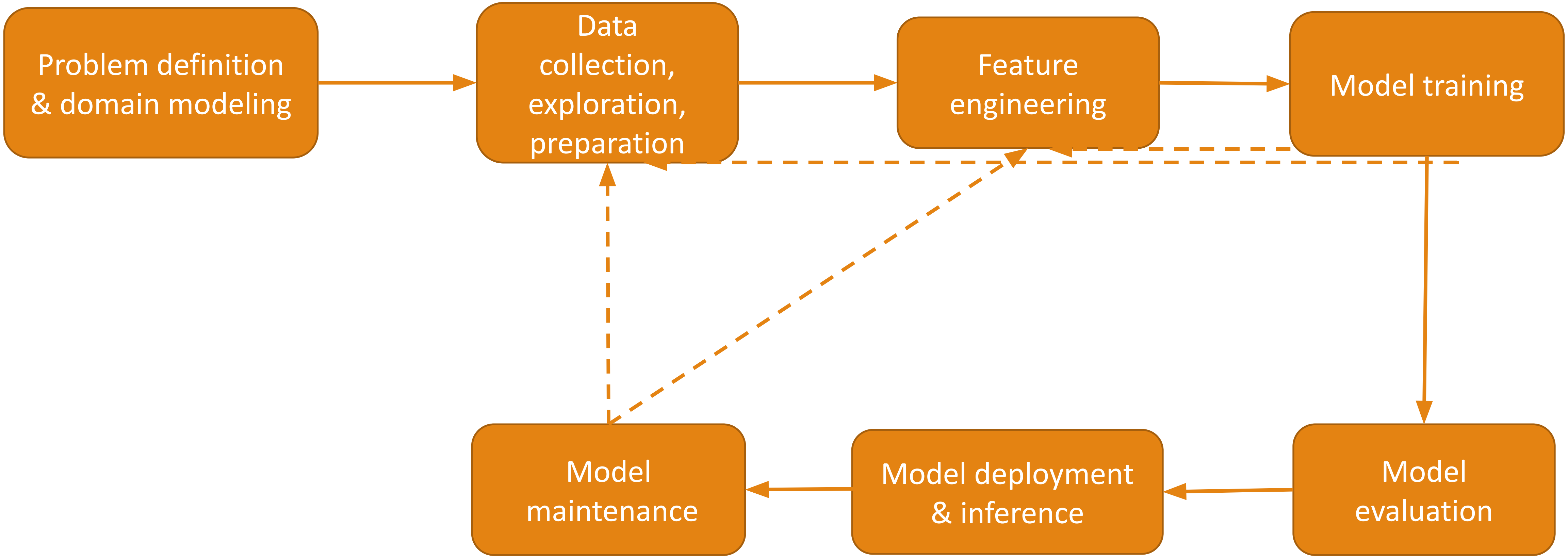
- What to learn from post-layout extracted netlists?
  - Leverage linear system theory to model POLO net approximately
    - Effective time constant
    - Total incident cap
- How to represent estimated interconnects?
  - Predict effective time constant and total incident cap
  - Use SPEF format to represent a "topology"
  - Topology:
    - Star vs delta network
    - Star network does not increase dense nodes
    - Delta network would increase dense nodes
  - The number of nodes increased is linear to the number of MOS devices
  - Simulation runtime increase is a modest 20%




$$\tau_{\mathrm{eff}} = \sqrt{\frac{1}{{p_1}^2} + \cdots + \frac{1}{{p_i}^2} + \cdots + \frac{1}{{p_N}^2}}$$

$$R_{eff} = \frac{\tau_{eff}}{C_{eff}}$$

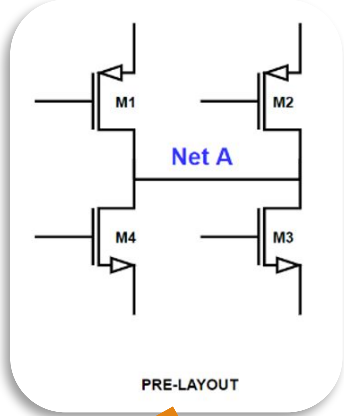
# Machine Learning Project Life Cycle

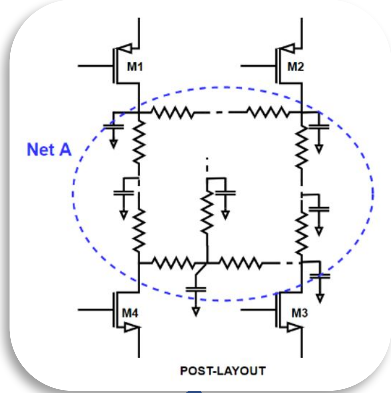



**Life Cycle of an ML Project [1]**

# Parasitic Estimation using ML

- A crawler to go through archived design database of pre and post layout netlists
- Gather attributes for each net in every circuit
  - *Features:*
    - *Number of MOS*
    - *Number of PMOS*
    - *Number of NMOS*
    - *Number of other devices*
    - *Width*
    - *Length*
    - *Number of ports*
    - *IO_PIN*
    - *Width of NMOS devices*
    - *Width of PMOS devices*
  - Output variables:
    - Total Net Capacitance
    - Time Constant/effective resistance




x

y

| #MOS | #PMOS | #NMOS | … | IO_PIN | … | # Ports | Total Net Cap |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 2 | … | 0 | … | 0 | 1.123 fF |

# Hands on ML: Step-by-Step Example

- Goal: Exposition of ML in EDA (MLParest)
- Does not cover:
  - Data collection
  - Resistance estimation
- Files: https://github.com/prateek-bhansali/parasitic_estimation_tutorial
  - Normalized input data for training
  - Jupyter Notebook

# Data Exploration

- Look at min, max, standard deviation, correlation, etc. of the dataset
  - Helps in flushing out bugs, modeling, figuring out new features, outliers, removing uncorrelated features

**Statistics of Numerical Data**

```
df.describe()
```

| | cap | f1 | f2 | f3 | f4 | f5 | f6 | f7 |
|---|---|---|---|---|---|---|---|---|
| **count** | 137505.000000 | 137505.000000 | 137505.000000 | 137505.000000 | 137505.000000 | 137505.000000 | 137505.000000 | 137505.000000 |
| **mean** | 0.007995 | 0.002204 | 0.080579 | 0.001112 | 0.000554 | 0.000711 | 0.001249 | 0.002838 |
| **std** | 0.025053 | 0.009111 | 0.272188 | 0.006975 | 0.005343 | 0.005152 | 0.011249 | 0.012596 |
| **min** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| **25%** | 0.000544 | 0.000710 | 0.000000 | 0.000177 | 0.000073 | 0.000103 | 0.000000 | 0.000512 |
| **50%** | 0.001488 | 0.000993 | 0.000000 | 0.000302 | 0.000146 | 0.000165 | 0.000000 | 0.001025 |
| **75%** | 0.006313 | 0.001561 | 0.000000 | 0.000781 | 0.000366 | 0.000413 | 0.000558 | 0.002049 |
| **max** | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

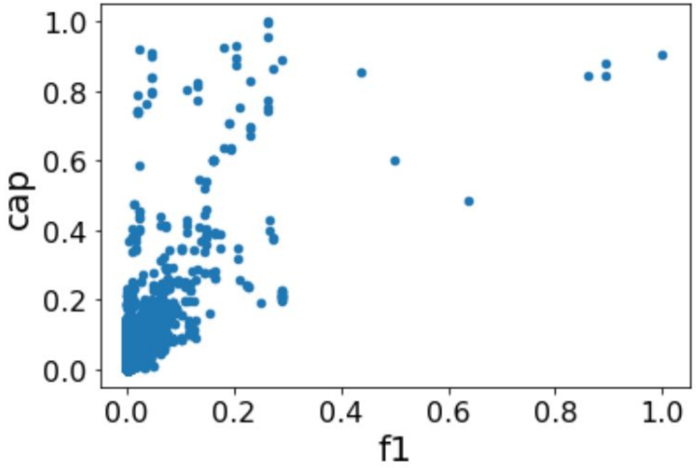
# Data Exploration: Correlation

```
corr_matrix = df.corr()
```

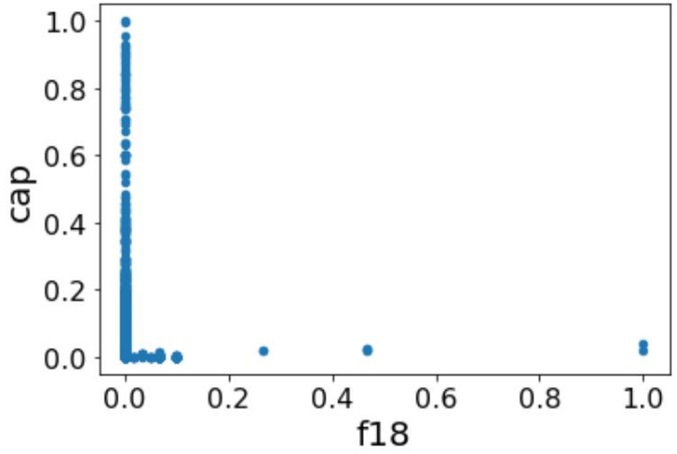
```
corr_matrix["cap"]
```

```
cap    1.000000
f1     0.622855
f2     0.111688
f3     0.539515
f4     0.364528
f5     0.423528
f6     0.323822
f7     0.455826
f8     0.423008
f9     0.455034
f10    0.038039
f11    0.422737
f12    0.479989
f13    0.542589
f14    0.421529
f15    0.311896
f16    0.396298
f17    0.003178
f18    0.000856
f19    0.028591
f20    0.620966
f21    0.613285
f22    0.562858
Name: cap, dtype: float64
```

```
ax = df.plot(kind="scatter", x="f1", y="cap")
```




```
ax = df.plot(kind="scatter", x="f18", y="cap")
```



# Data Preparation

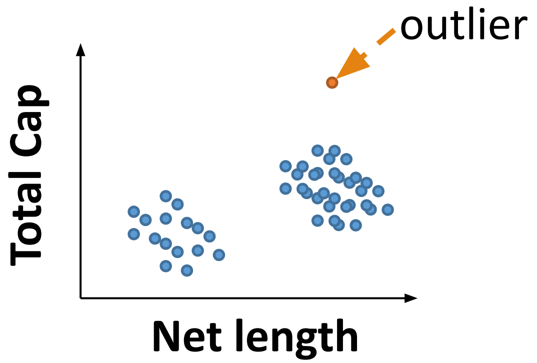
- Identify target and input features

## Prepare the Data For Machine Learning Algorithms

```
target_cols = ['cap']

# numerical attributes
num_attribs = ['f1','f2','f3','f4', 'f5', 'f6', 'f7', 'f8', 'f9', 'f10', 'f11', 'f12', 'f13', 'f14', 'f15', 'f16', 'f17', 'f18', 'f19',
       'f20', 'f21', 'f22']

# categorical attributes (we do not have any in this case)
cat_attribs = []

# get the meaningful attributes and target_cols
circuit = df[num_attribs+cat_attribs+target_cols].copy()
```

# Data Preparation

- Outlier detection
  - Helps in eliminating dirty nets from opens/shorts
  - Used RANSAC algorithm
- Normalization
  - Different features are on different scale: w, l, number of MOS
  - Min-max scaling or standard scaler scaling



# Data Preparation: Outlier Detection

## Outliner Detection: Use RANSAC (RANdom SAmple Consensus) algorithm.

```
def filter_outliers(X, Y, max_trials=500, sigma_scale=3):

    X_d = X.copy()
    Y_d = Y.copy()
    print("Length of Original data: ", len(X_d))
    min_samples = np.floor(len(X_d)/2)
    lmr = linear_model.RANSACRegressor(base_estimator=linear_model.LinearRegression(copy_X=True, normalize=True),
                                       min_samples=min_samples, residual_threshold=sigma_scale*np.std([Y]), max_trials=max_trials, random_state=137)
    model = lmr.fit(X_d, Y)
    inlier_mask = lmr.inlier_mask_
    outlier_mask = np.logical_not(inlier_mask)

    X = X_d[inlier_mask].copy()
    Y = Y_d[inlier_mask].copy()

    print("Length of learing data: ", len(X))
    print("Percentage of original data: ", len(X)/len(X_d))
    print("Number of outliers found: ", len(X_d)-len(X))

    return X, Y
X = circuit[num_attribs+cat_attribs]
Y = circuit[target_cols]
X, Y = filter_outliers(X, Y)
```

```
Length of Original data:  137505
Length of learing data:  136607
Percentage of original data:  0.9934693283880586
Number of outliers found:  898
```

# Feature Engineering

- **Option A:** Manually engineer non-linear features and use them in a linear regression ML model
  - Requires human resources and domain knowledge
- **Option B:** Use inherently non-linear models like Random Forest(RF), Gradient Boosted Decision Trees (GBDT) or Neural Network (NN)
  - Does not require human intervention
  - RF led to great results

| | cap | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | ... | f13 | f14 | f15 | f16 | f17 | f18 | f19 | f20 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.077530 | 0.023840 | 1 | 0.014869 | 0.006585 | 0.009917 | 0.0 | 0.000000 | 0.000000 | 0.016164 | ... | 0.025862 | 0.000000 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.023698 | 0 |
| 1 | 0.022282 | 0.007805 | 0 | 0.001952 | 0.000878 | 0.001322 | 0.0 | 0.003586 | 0.001870 | 0.001616 | ... | 0.006466 | 0.009021 | 0.0 | 0.004187 | 0.0 | 0.0 | 0.0 | 0.007805 | 0. |

# Train/Validation/Test Split

- Traditionally, data is split in train (80%), validation (10%) and testing(10%) sets
  - Labeled data is not massive, so we only do training/testing split in MLParest
  - Use K-Fold Cross Validation (CV) for tuning parameters
  - Can we split based on circuits instead?
    - 80% of circuits (and their nets) are used for training and rest 20% are used for testing

# Data Normalization and One-hot Encoding using Pipelines

## Transformation Pipelines

```
# numerical pipeline object -- you may do data imputation here
#  -- we will do standard scaler transformation to our data
num_pipeline = Pipeline([('std_scaler', StandardScaler())])

# full pipeline with one-hot encoding of categorical inputs
full_pipeline = ColumnTransformer([
                ("num", num_pipeline, num_attribs),
                ("cat", OneHotEncoder(), cat_attribs),
                ])
X_train_prepared = full_pipeline.fit_transform(X_train)
```

# Model Training

- Selection of model: factor in complexity, inference/training time, data volume, interpretability, fitting…
- We found RF/GBDT to be robust to overfitting in practice
- Training time was not a concern as amount of data is not massive

| Model Type | Training Speed | Training Data needed | Inference Speed | Accuracy |
|---|---|---|---|---|
| Linear Models | Fastest | Low | Fastest | Low |
| Ensemble Methods (GBDT, RF) | Fast (RF can be parallelized) | Moderate | Fast | Great |
| Neural Networks | Slow | High | Slower | Best |

# Model Evaluation Metrics

- How to check if model is performing well?
  - Offline/Batch Metric: a proxy for simulation accuracy
    - Classification tasks: MLParest is not classification.
      - Precision, recall, F1 score
    - **Regression: We use RMSE for MLParest**
      - Root Mean Square Error (RMSE) or Mean Absolute Error (MAE)?
      - Try both to see what works best for your application
      - Sci-kit has non-optimal implementation of MAE as of June 2021 – runs slow.
  - Online Metric:
    - Simulation accuracy: available when SPICE simulations are run with post-layout data

# Default Linear, RF and GBDT Accuracy

### Default Linear Regression Model

```
%%time
# Ordinary least squares Linear Regression.
"""
LinearRegression fits a linear model with coefficients w = (w1, ..., wp)
to minimize the residual sum of squares between the observed targets in
the dataset, and the targets predicted by the linear approximation.
"""

lin_reg = LinearRegression()
lin_reg.fit(X_train_prepared, Y_train)
print("RMSE of default Linear Model: ", get_model_rmse(lin_reg, full_pipeline,  X_train, Y_train))
```

```
RMSE of default Linear Model:  0.012109838428412388
CPU times: user 284 ms, sys: 271 ms, total: 556 ms
Wall time: 165 ms
```

### Default Random Forest (RF) Regression

```
%%time
# Random Forest Regressor
"""
A random forest is a meta estimator that fits a number of classifying
decision trees on various sub-samples of the dataset and uses averaging
to improve the predictive accuracy and control over-fitting.
"""
forest_reg = RandomForestRegressor(n_jobs=-1)
forest_reg.fit(X_train_prepared, Y_train.values)
print("RMSE of default RF Model: ",get_model_rmse(forest_reg, full_pipeline,  X_train, Y_train))
```

```
RMSE of default RF Model:  0.0033786973865209393
CPU times: user 30 s, sys: 295 ms, total: 30.3 s
Wall time: 7.78 s
```

### Default Gradient Boosted Decision Trees (GBDT)

```
%%time
#Gradient Boosting for regression.
"""
GB builds an additive model in a forward stage-wise fashion;
"""
gbdt_reg = GradientBoostingRegressor(random_state=0)
gbdt_reg.fit(X_train_prepared, Y_train.values)
print("RMSE of default GBDT Model: ",get_model_rmse(gbdt_reg, full_pipeline,  X_train, Y_train))
```

```
RMSE of default GBDT Model:  0.006332032559214701
CPU times: user 10.9 s, sys: 0 ns, total: 10.9 s
Wall time: 10.9 s
```

| Model (default parameters) | Accuracy (RMSE) | Time |
|---|---|---|
| Linear | low | low |
| GDBT | medium | high |
| RF | high | medium |

# Cross Validation Scores of RF and GBDT

### CV Score of Linear Regression

```
scores = cross_val_score(lin_reg, X_train_prepared, Y_train, scoring = "neg_mean_squared_error", cv=5)
lin_rmse_scores = np.sqrt(-scores)
display_cv_scores(lin_rmse_scores)
```

```
Scores: [0.01328559 0.01195765 0.0121923  0.01214132 0.01212787]
Mean: 0.012340946430903676
Standard deviation: 0.00047887469420691375
```

### CV Score of Random Forest Regression

```
scores = cross_val_score(forest_reg, X_train_prepared, Y_train, scoring = "neg_mean_squared_error", cv=5)
forest_rmse_scores = np.sqrt(-scores)
display_cv_scores(forest_rmse_scores)
```

```
Scores: [0.00406476 0.00492766 0.00536735 0.00400481 0.00380739]
Mean: 0.0044343943083576965
Standard deviation: 0.0006046536760889458
```

### CV Score of GBDT

```
scores = cross_val_score(gbdt_reg, X_train_prepared, Y_train, scoring = "neg_mean_squared_error", cv=5)
gbdt_rmse_scores = np.sqrt(-scores)
display_cv_scores(gbdt_rmse_scores)
```

```
Scores: [0.00669476 0.00684345 0.00741301 0.00654892 0.00647032]
Mean: 0.006794091938922407
Standard deviation: 0.00033475310988066275
```

| Model (default parameters) | CV Score (lower is better) |
|---|---|
| Linear | High |
| GDBT | Low |
| RF | Low |

# Hyperparameter Tuning

```python
%%time

# Define a parameter grid and do hyperparameter tuning for RF model

"""
param_grid = [
    {'n_estimators': [100], 'max_depth':[10,15,None]},
    ]
"""
param_grid = [
    {'n_estimators': [50, 100, 150, 200],
     'max_depth':[10,15,None],
     'max_features': ["auto", "sqrt", "log2", None],
     'min_samples_leaf':[1, 5, 10, 15, 20]}
    ]
forest_reg_gs = RandomForestRegressor(n_jobs=-1)

grid_search_forest = HalvingGridSearchCV(forest_reg_gs, param_grid, cv=5,
                                  scoring='neg_mean_squared_error',
                                  return_train_score=True, verbose=0, n_jobs=-1)

grid_search_forest.fit(X_train_prepared, Y_train)
print("Best RF estimator", grid_search_forest.best_estimator_)
print("RMSE of tuned RF model is :", get_model_rmse(grid_search_forest, full_pipeline, X_train, Y_train))
```

```
Best RF estimator RandomForestRegressor(max_features='sqrt', n_estimators=200, n_jobs=-1)
RMSE of tuned RF model is : 0.0033875927206198746
CPU times: user 27.1 s, sys: 270 ms, total: 27.4 s
Wall time: 6min 24s
```

```python
%%time
# Define a parameter grid and do hyperparameter tuning for GBDT model
"""
param_grid = [
    {'n_estimators': [100], 'max_depth':[10,15,None]},
    ]
"""
param_grid = [
    {'n_estimators': [50, 100, 150, 200],
     'max_depth':[10,15,None],
     'max_features': ["auto", "sqrt", "log2", None],
     'min_samples_leaf':[1, 5, 10, 15, 20]}
    ]
gbdt_reg_gs = GradientBoostingRegressor()

grid_search_gbdt = HalvingGridSearchCV(gbdt_reg_gs, param_grid, cv=5,
                                  scoring='neg_mean_squared_error',
                                  return_train_score=True, verbose=0, n_jobs=-1)

grid_search_gbdt.fit(X_train_prepared, Y_train)
print("Best GBDT estimator", grid_search_gbdt.best_estimator_)
print("RMSE of tuned GBDT model is ", get_model_rmse(grid_search_gbdt, full_pipeline, X_train, Y_train))
```

```
Best GBDT estimator GradientBoostingRegressor(max_depth=10, max_features='sqrt')
RMSE of tuned GBDT model is  0.003281195598028159
CPU times: user 15 s, sys: 780 ms, total: 15.8 s
Wall time: 4min 46s
```

| Model | CV Score Improved? |
|---|---|
| GDBT | Yes |
| RF | Yes |

# Quick Testing of Fitted Models

```
# plotting
some_data = X_train.iloc[:5]
some_labels = Y_train.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)
print("Predictions       RF Model:", forest_reg.predict(some_data_prepared))
print("Predictions tuned RF Model:", grid_search_forest.predict(some_data_prepared))
print("Predictions       GBDT Model", gbdt_reg.predict(some_data_prepared))
print("Predictions tuned GBDT Model", grid_search_gbdt.predict(some_data_prepared))
print("Labels:      ", some_labels.transpose())
```

```
Predictions       RF Model: [0.00286248 0.00046561 0.00305074 0.00048138 0.07553159]
Predictions tuned RF Model: [0.00284831 0.00046582 0.00304563 0.00048168 0.0755103 ]
Predictions       GBDT Model [0.00945122 0.00063459 0.00372003 0.00144793 0.07204102]
Predictions tuned GBDT Model [0.00336014 0.00045715 0.00280984 0.00049241 0.07551923]
```

| | | | | | |
|---|---|---|---|---|---|
| Actual Cap | 0.003023 | 0.000565 | 0.002592 | 0.000482 | 0.081947 |

# RF and GBDT on Testing Data

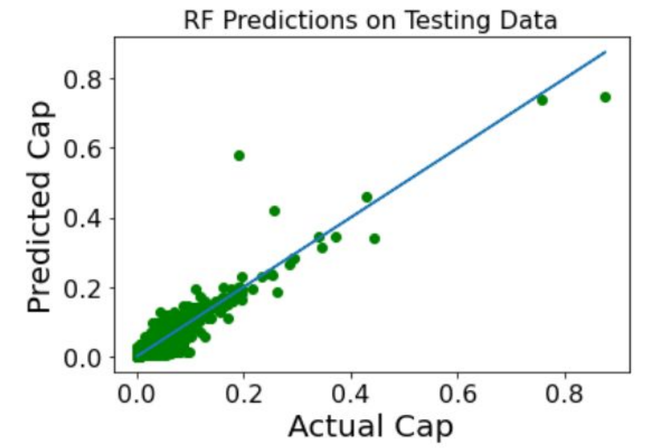
## Final Test

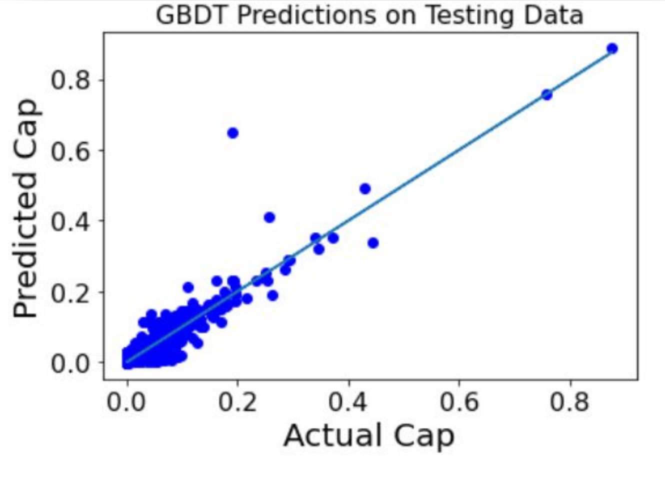
```
# Final Test
### Random Forest Metrics on Test Data
final_model = grid_search_forest.best_estimator_
X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)
R2 = r2_score(Y_test, final_predictions)
print("R2 Score is : ", R2)
print("RMSE is:", get_model_rmse(grid_search_forest.best_estimator_, full_pipeline, X_test, Y_test))
```

```
R2 Score is :  0.9388278509745729
RMSE is: 0.006805427527653161
```

```
### GBDT Metrics on Test Data
final_model = grid_search_gbdt.best_estimator_
X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)
R2 = r2_score(Y_test, final_predictions)
print("R2 Score is : ", R2)
print("RMSE is:", get_model_rmse(grid_search_gbdt.best_estimator_, full_pipeline, X_test, Y_test))
```

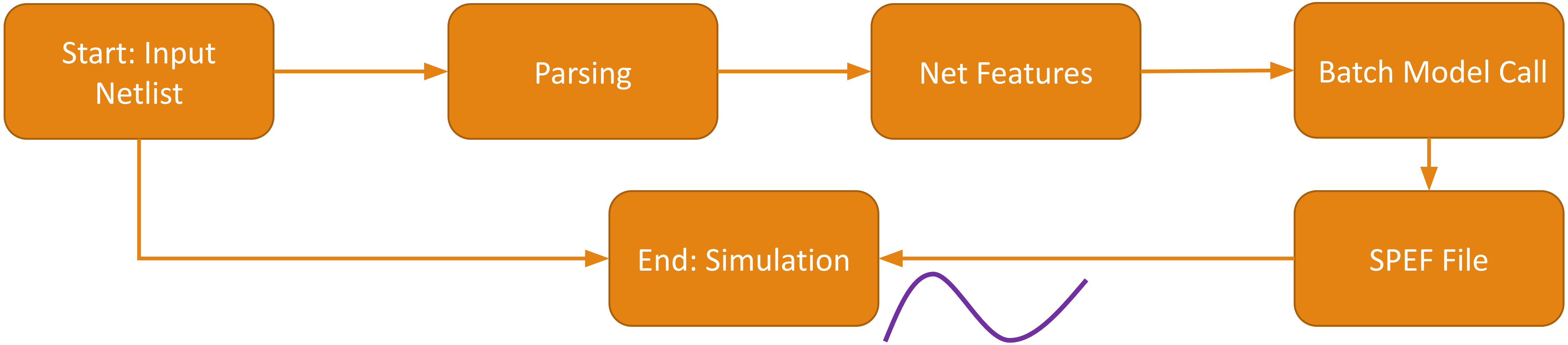
```
R2 Score is :  0.8415378355758799
RMSE is: 0.0069953483921755
```



# Model Inference

- Parse the netlist and generate net features
- Do a batch call to save time in inference
- Tweak/bound predicted values based on domain knowledge
- Generate SPEF file
- Monitor any errors/NaNs



# Model Deployment

- Model can be deployed on cloud or distributed file system like HDFS/NFS
  - If you deploy on a cloud, you can send HTTP request and get response
  - Use caching to reduce network calls
  - Save hyperparameters as part of the model or do some sort of versioning
  - Reduce model size by pruning redundant branches/features
    - May be needed to reduce peak memory consumption in the flow

# References

[1] “Machine Learning Engineering”, Andriy Burkov

[2] B. Shook, P. Bhansali, C. Kashyap, C. Amin and S. Joshi, "MLParest: Machine Learning based Parasitic Estimation for Custom Circuit Design," *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1-6, doi: 10.1109/DAC18072.2020.9218495.