CS 181 - Machine Learning (Spring 2016)
Practical 1: Predicting the Efficiency of Organic Photovoltaics

Team BridgingtheGAK

Gioia Dominedò (40966234) Amy Lee (60984077) Kendrick Lo (70984997)

February 12, 2016

For this project we used machine learning methods to predict the potential efficiency of different molecules as building blocks for solar cells, based on molecular structure features encoded in the form of SMILES strings. More specifically, we produced continuous real-valued predictions of the "gap" that would otherwise be estimated using density functional theory (*DFT*), where the gap represents the difference in energy between the highest occupied molecular orbital (*HOMO*) and the lowest unoccupied molecular orbital (*LUMO*).

At the time of submission, we had uploaded 23 different sets of predictions to Kaggle and achieved a best root mean squared error (*RMSE*) of 0.053 on the public test data, corresponding to a ranking of 3/86 on the leaderboard. This is subject to change, pending release of the withheld final test data. Our best results were achieved using a tuned random forest model that used a combination of features, including features from the original dataset, RDKit-generated features associated with molecular bond counts, and counts of contiguous character sequences within the SMILES strings.

1 Technical Approach

The primary modeling techniques used were: **linear regression** and **random forests**. We also spent a considerable amount of time on feature engineering, which included both character string analysis and use of RDKit. Our full code base can be found in our process books, which are described in more detail in Appendix C.

1.1 Exploratory Data Analysis

We performed a basic linear regression in R and explored, among other things, the distribution of the response variable and whether the assumptions of linear regression (independence, normality, constant variance, linearity) were likely to be satisfied. We analyzed summary statistics and a number of plots associated with this preliminary linear regression model (e.g. Figures 1 and 2). In addition to highlighting some heteroscedasticity, our visual analysis showed the presence of three **outliers**. Upon further investigation, we confirmed that these outliers were associated with the only three (out of 1M) entries that had a negative HOMO-LUMO gap value; whether these are valid values or not should be confirmed by a domain expert.

1.2 Pre-processing

We performed a backward selection technique and discovered that all but 31 columns in the training set were zero vectors; therefore, we reduced the dimensionality of our data set by removing these columns since they would not add any additional information to our model (due to lack of variability). There was no need to standardize these features as they were all indicator variables; however, we considered that we might need to standardize other numerical features that we would generate in later stages, depending on the models used (n.b. random forest models do not require standardization).

1.3 Linear Regression

We fit linear regression models to various design matrices, using both the features from the original data set and additional interaction terms. We also tried a number of **regularization** techniques, which add regularization terms to models' loss functions in order to penalize complexity (i.e. larger coefficients) and minimize in-sample overfitting. More specifically, we tested: *ridge regression*, which uses a L2 norm penalty of the form $\lambda \sum w_j^2$; *LASSO*, which uses a L1 norm penalty of the form $\lambda \sum |w_j|$; and *Elastic Net*, which combines the L1 and L2 norms. It should be noted that a key difference between the three techniques is that ridge regression typically results in dense solutions where most coefficients are non-zero, while LASSO and Elastic Net leads to sparse solutions, where most coefficients are driven to zero.

We tested these three regularization techniques with different λ parameters for several models with different feature combinations. However, we found that the linear regression models consistently underperformed the random forest models — suggesting that they were not sufficiently expressive to accurately predict the HOMO-LUMO gap — so we focused our tuning efforts (and the discussion in this report) on the latter model category.

1.4 Random Forest Regression

Unlike linear regression, random forests are a non-parametric model class. As with other ensemble learning methods, random forest models tend to do very well in model fitting where accuracy of fit (and predictive power) is prioritized over model interpretability. While decision trees typically over-fit the training set, random forest models reduce variance by training on subsets of both the training data and the design features and, consequently, usually generalize better to out-of-sample data. As we began to see them significantly outperform the linear regression models, we focused on this class of models in later stages of the project. Our parameter tuning is described in more detail below.

1.5 Validation

We considered performing cross-validation in our modeling to avoid over-fitting; however, given time constraints and the large volume of training data, we instead chose to set aside a portion of the training data for use as a validation set (or "test set" in the context of our training set). This not only allowed us to check how well our models generalized before submitting them to Kaggle, but it also allowed us to compare performance across models on a consistent test-train split. We used a common 80-20 split for this purpose.

1.6 Feature Engineering

Most importantly, we found that adding new (judiciously selected) features provided the biggest gains in model performance. Accordingly, after our initial exploration of the original data, we set out to generate new features for infusion into our models. We experienced success with two primary approaches: **generating features via RDKit** and **character string analysis**.

1.6.1 RDKit

We discovered useful domain knowledge and meaningful feature generating functionality by exploring the documentation and experimenting with coding in RDKit[1], an open-source chemical informatics software. First, we discovered that the 256 features provided in the original dataset were molecular fingerprints, where each bit indicates the presence or absence of particular chemical substructures in the molecule. We experimented with generating 2048-bit fingerprints on a sample of 50,000 molecules and found that performance improved when moving from the 256 features provided, to the 2,048 newly generated features. However, we were not able to pursue this further on the entire dataset due to memory and time constraints.

We also experimented with RDKit functionality that allowed us to identify different types of atoms and bonds from the parsed SMILES strings. Since electrons play a major role in chemical bonds and bonds communicate information about how atoms interact, we thought to generate new features by counting the total number of each type of chemical bond present. We found that there were only three types of bonds in the entire dataset: aromatic, double, and single. Augmenting the data with just these three features allowed us to improve the random forest RMSE from the baseline of 0.27 to 0.18 without tuning any parameters.

Although we did not continue to explore the 2048-bit features, we discovered that we could use them to calculate the similarities between any two molecules and to query substructures. We selected 200 representative molecules from the test set that were evenly spaced across the full range of HOMO-LUMO gaps and calculated the Tanimoto Coefficient (a similarity ratio over binary attributes) between each molecule and these 200 representatives. For fun, we also used the SMILES string for chlorophyll as representative number 201, given that it is an bio-organic compound known for its role in photosynthesis. Using a random forest model with default parameters, the 200 (+1) similarity features alone without any other features achieved a 0.15 RMSE on Kaggle.

1.6.2 Character String Analysis

As a complement to the feature creation using RDKit, we also directly parsed the SMILES strings in order to identify contiguous character sequences. For example, a one-character sequence might include C and a two-character sequence might include Si (i.e. a single atom) or CO (i.e. two adjacent atoms). By extending to increasingly longer *character* sequences, we are effectively looking at longer *chemical* sequences; that is, we are considering not only the number of individual chemical structures but also the order in which they appear.

[1] http://www.rdkit.org/

We calculated a count of unique n-character features on a scrolling basis[2]. We started our performance testing using a random forest model with default parameters. Our first model included all unique one-character sequences and the original non-zero binary features, and resulted in a substantial drop in RMSE (0.103 training; 0.142 validation). The RMSE improved even further when moving to two-character sequences (0.051 training; 0.100 validation) and three-character sequences (0.033 training; 0.073 validation; 0.068 Kaggle), indicating that we were successfully reducing bias without introducing variance. At this point, we tried adding the three bond type features from RDKit; this resulted in further improvements, though on a smaller scale (0.031 training; 0.067 validation; 0.062 Kaggle). Due to the amount of memory required to generate additional features, we were not able to work with longer character sequences.

While performance improved incrementally when moving to longer character sequences, we found that aggregating features based on sequences of different lengths resulted in over-fitting. For example, when combining the one-, two-, and three-character sequences, the RMSE on the validation set increased from 0.067 to 0.072. For this reason, we chose to move on to tuning the model with the three-character, RDKit bond type, and original non-zero binary features, which had moved us into a top-five spot on the Kaggle leaderboard just by using default random forest parameters.

1.7 Parameter Tuning

Our testing revealed that further incremental gains could be made by tuning the parameters of our "best features" Random Forest model. We found that increasing the number of trees built before taking the prediction average (*n_estimators*) resulted in lower validation RMSEs, though at the cost of higher computing times. This was in line with our expectations: by design, random forests are intended to minimize overfit and this positive effect is amplified as the number of trees grows. We also found that limiting the number of features considered when choosing the best split (*max_features*) resulted in lower RMSEs on the validation set due to reduced variance, though this improvement was eroded by an increase in bias when choosing too small a subset of features. Table 1 shows this relationship in detail; we found that the optimal parameter varied between 0.6 and 0.8, depending on the number of trees. Finally, we tested the minimum number of samples in newly created leaves (*min_samples_leaf*) and among those tested found the optimal parameter to be 1. We can explain this performance as follows: allowing for a small number of samples in individual leaves permits the model to more accurately fit the target variable. While this can make a model prone to overfitting, we posit that the risk is reduced in this case as the target variable is based on DFT calculations rather than observations that could include measurement error.

2 Results

We had uploaded 23 sets of predictions to Kaggle at the time of submission. We selected the models to be tested on Kaggle based on their performance on the validation set that we carved out from the training data; however, we generated submissions based on models trained on the entire training set in order to make full use of the training data available.

[2]For example, 'ABCD' has four one-character features ('A', 'B', 'C', 'D'), three two-character features ('AB', 'BC', 'CD'), two three-character features ('ABC', 'BCD') and one four-character feature ('ABCD').

The best-performing models not only exceeded the baselines but also performed excellently overall, allowing us to reach a ranking of 3/86 on the leaderboard at the time of submission (though the performance of our models on the withheld data remains to be seen). Table 2 summarizes the performance of the models that we built, including their RMSEs on the training and validation sets, and on Kaggle if submitted.

We selected the following two models as our "best" submissions for the Kaggle competition:

- **Features:** 3-character sequences; bond type features; original non-zero binary features
  Model: Random forest; 150 trees; $0.6 * n$ maximum features; 1 minimum sample size per leaf
- **Features:** 3-character sequences; bond type features; original non-zero binary features
  Model: Random forest; 150 trees; $0.7 * n$ maximum features; 1 minimum sample size per leaf

We note that the RMSE on the validation set was consistently higher than the RMSE on the training set, indicating a certain degree of overfitting to the training data and emphasizing the importance of using validation (or cross-validation) to obtain a more accurate metric of performance on unseen data. The importance of the validation set was further confirmed by the fact that model performance of Kaggle was consistently in line with that observed on the validation set, indicating that our models generalized well to the unseen data despite some degree of overfitting on the training data. The performance was no doubt helped by the large volume of data; for smaller datasets, it is possible that our models would not have generalized as well and that cross-validation would have been required.

3 Discussion

The discussion above details our thought process and the techniques that we employed through the project. Our main takeaways are listed below.

1. **Importance of feature selection**: We noted early in the process that parameter tuning only resulted in marginal incremental gains in performance, while adding descriptive features resulted in significant drops in RMSE. We tried to be as creative as possible when constructing new features, and also performed some domain research to guide our efforts (none of us have a chemistry background).
2. **Targeted parameter tuning**: Tuning our random forest model highlighted the fact that the optimal parameters are driven by the nature of the prediction problem and data. For example, we found that performance was optimized when using `in_samples_leaf = 1`; while this low a value can make a model prone to overfitting, we believe that this is not happening in this case because the target variable is based on DFT calculations instead of noisy observations.
3. **In-sample vs out-of-sample performance**: The differences between the training set and validation set RMSE highlighted the importance of validation (or cross-validation) when building predictive models that generalize well to out-of-sample data. Using a training-validation split allowed us to obtain a metric of performance on unseen data and minimized the risk of overfitting.

A Figures

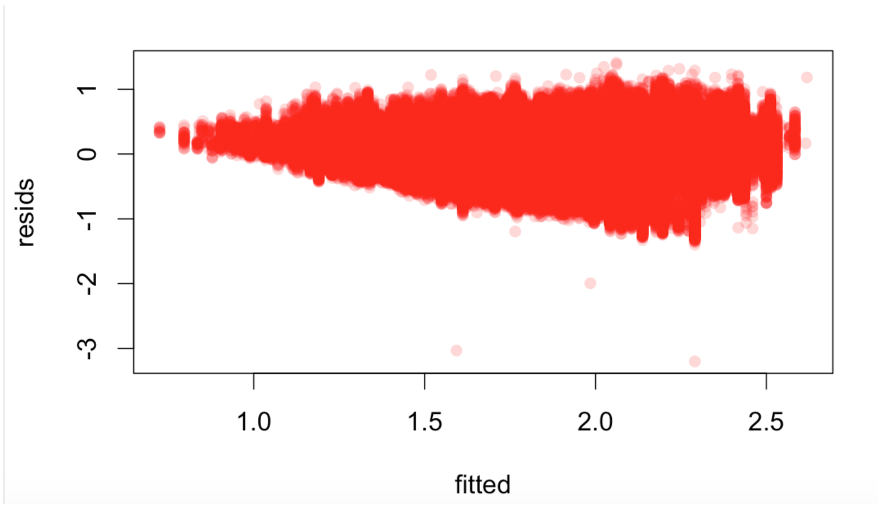

Figure 1: Plot of Residuals versus Fitted Values on Baseline Linear Regression

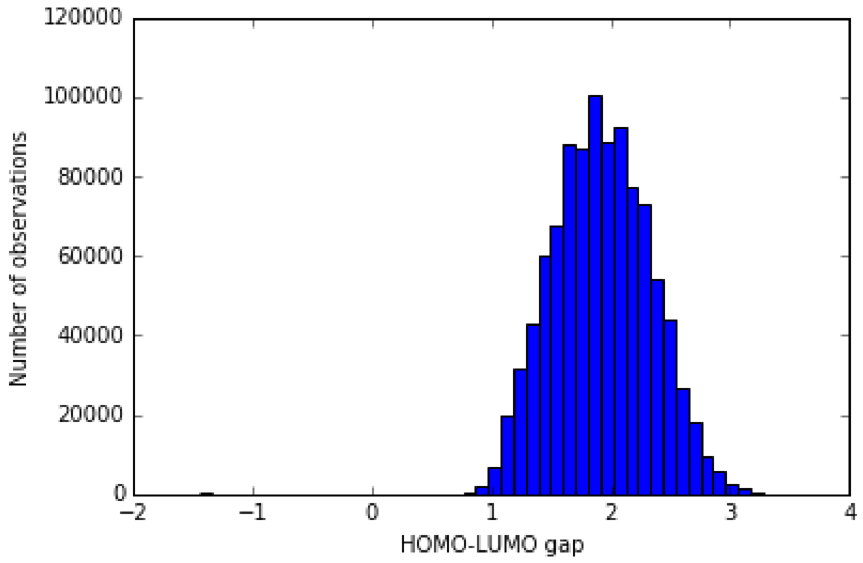

Figure 2: Target Variable Distribution

B Model Performance Summary

Table 1: "Best Features" Random Forest Model Parameter Tuning

	Number of Trees				
Maximum Features	10	20	50	100	150
$\sqrt{n}$	0.080	0.078	0.076	0.074	-
$0.3 * n$	0.067	0.063	0.060	0.059	-
$0.5 * n$	0.065	0.061	0.059	0.059	-
$0.6 * n$	0.065	0.061	0.058	0.058	0.058
$0.7 * n$	0.063	0.060	0.058	0.058	0.064
$0.8 * n$	0.064	0.061	0.056	0.059	-
$0.9 * n$	0.064	0.062	0.060	0.060	-
$1.0 * n$	0.067	0.064	0.063	0.062	-

(Validation Set RMSE)

Table 2: Root-Mean-Squared Errors

Attempt	Class	Subclass	Description	Train	Validation	Kaggle
33	RF		tuned 3-char. + bond count (best)	0.024	0.058	0.053
32	RF		tuned 3-character	0.026	0.064	0.059
31	RF		3-char. + bond count + Tanimoto	0.043	0.098	-
30	RF		1-3-character subsequences	0.033	0.072	-
29	RF		3-character + bond count	0.030	0.067	0.062
28	RF		3-character subsequences	0.033	0.073	0.068
27	RF		2-character subsequences	0.051	0.100	-
26	RF		1-character subsequences	0.103	0.141	-
25	RF		Tanimoto + bonds + 256-bit	0.055	0.126	-
24	RF		Tanimoto Similarity (201 molecules)	0.071	-	0.154
23	RF		bonds + elements + interactions	0.098	0.149	-
22	LR		bonds + elements + interactions	0.195	0.195	-
21	RF		bond count + element symbols	0.118	0.158	-
20	RF		add RDKit bond count	0.184	-	0.186
19	RF		count element symbols	0.118	0.158	-
18	LR	lasso	count element symbols	0.243	0.243	-
17	LR		count element symbols	0.243	0.243	-
16	RF		2048-bit fingerprint	0.071	0.162	-
15	LR	lasso	2048-bit fingerprint	0.137	0.144	-
14	LR		2048-bit fingerprint	0.136	0.144	-
13	RF		transform response	0.272	0.274	0.272
12	LR		transform response	0.280	0.281	-
11	LR	EN	as below, tuned parameters	0.280	0.281	-
10	LR	ridge	as below, tuned parameters	0.280	0.281	-
9	LR	lasso	as below, tuned parameters	0.281	0.282	-
8	RF		reduced set (28) + interactions	0.272	0.273	0.272
7	LR		reduced set (28) + interactions	0.280	0.281	-
6	RF		add pairwise interactions	0.272	0.273	-
5	LR		add pairwise interactions	0.280	0.281	-
4	RF		reduced set (31) + sum of variables	0.272	0.273	-
3	LR		reduced set (31) + sum of variables	0.299	0.300	-
2	RF		reduced set (31)	0.272	0.273	-
1	LR		reduced set (31)	0.299	0.300	-

(LR: Linear Regression; RF: Random Forest)

C Code Base

Details of the investigations we performed can be found in the following process books, accessible on Github[3] after the close of competition.

1. *practical1.Rmd*: Exploratory data analysis and feature reduction
2. *kl_initial_model_building.ipynb*: Initial model building (original features)
3. *gd1_256_binary_features.ipynb*: Further model building (original features)
4. *gd2_smiles_[feature description].ipynb (multiple notebooks)*: Feature engineering based on contiguous character sequences in SMILES strings
5. *al_bondtypes_counts.ipynb*: Feature engineering based on bond types counts generated with RDKit
6. *al_similarity.ipynb*: Feature engineering based on Tanimoto Coefficient similarities generated with RDKit
7. *al_playwithrdkit_gen-2048bit-fingerprint.ipynb*: Feature engineering based on 2048-bit molecular fingerprints generated with RDKit

We have included below the code that was used to generate our best-performing random forest model.

C.1 Generate Three-Character Sequence Counts

```
import time
import numpy as np
import math
import pandas as pd
import itertools

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

# load all data
df_train = pd.read_csv('train.csv')
df_test = pd.read_csv('test.csv')

# store gap values
Y_train = df_train.gap.values

# row where testing examples start
test_idx = df_train.shape[0]

# delete 'Id' column
df_test = df_test.drop(['Id'], axis=1)

# delete 'gap' column
df_train = df_train.drop(['gap'], axis=1)
```

[3]https://github.com/dominedo/kaggle-competitions/tree/master/1-photovoltaic-molecules

```
# check original dataframe sizes
df_train.shape, df_test.shape

# dataframe with all train and test examples so we can more easily apply feature engineering
df_all = pd.concat((df_train, df_test), axis=0)

# extract 'smiles' column - for separate processing
df_smiles = pd.DataFrame(df_all.smiles.values, columns=['smiles'], index=df_all.index.values)

# drop the 'smiles' column
df_all = df_all.drop(['smiles'], axis=1)

# drop columns that are all zeros
df_all_sum = pd.DataFrame(df_all.sum(axis=0), index=df_all.columns, columns=['SUM'])
df_all_zeros = df_all_sum[df_all_sum.SUM==0].index.values
df_all.drop(df_all_zeros, axis=1, inplace=True)

# number of original features
num_orig_features = df_all.shape[1]

def split_smiles(smile, chars):
    result = []
    for s in range(len(smile)-chars+1):
        result.append(smile[s:s+chars])
    return result

# create all string combinations
i = 3 # max=63
df_tmp = df_smiles.copy()
df_tmp = df_tmp.applymap(lambda x: split_smiles(x, i))
df_smiles['SEQ_' + str(i) + '_CHARS'] = df_tmp.smiles.copy()
unique_feat = sorted(set(list(itertools.chain(*df_smiles['SEQ_' + str(i) + '_CHARS'].values))))

# turn into features
i = 3 # max=63
for f in unique_feat:
    df_tmp = df_smiles['SEQ_' + str(i) + '_CHARS'].copy()
    df_tmp = df_tmp.map(lambda x: x.count(f))
    df_smiles['SEQ_' + str(i) + '_CHARS_' + f] = df_tmp

# combine old and new features
df_enh = pd.concat([df_smiles, df_all], axis=1)

# drop the 'smiles' column
df_enh = df_enh.drop(['smiles'], axis=1)

# drop the list columns
i = 3 # max=63
df_enh = df_enh.drop(['SEQ_' + str(i) + '_CHARS'], axis=1)

# see what we ended up with!
df_enh.shape

# split back up into training and test data
lcols = df_enh.columns.values.tolist()
vals = df_enh.values
X_train = vals[:test_idx]
X_test = vals[test_idx:]
print 'Train features:', X_train.shape
print 'Train gap:', Y_train.shape
print 'Test features:', X_test.shape

# tuning parameters
# http://www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/
# n_estimators : integer, optional (default=10)
num_trees = [10, 20, 50]

# max_features : int, float, string or None, optional (default=?auto?)
num_feat_consider_split = ["sqrt", "log2", 0.2, 0.3, 0.5, 0.6, 0.7, "auto"]

# min_samples_leaf : integer, optional (default=1)
min_samples_in_leaf = [1, 2, 3, 5, 10]

# random forest regressor - test & validation split
for a in num_trees:
    for b in num_feat_consider_split:
        for c in min_samples_in_leaf:
            RF = RandomForestRegressor(n_estimators=a, max_features=b, min_samples_leaf=c)
            RF.fit(X_train[:800000], Y_train[:800000])
            RF_rmse_train = math.sqrt(mean_squared_error(Y_train[:800000], RF.predict(X_train[:800000])))
            RF_rmse_val = math.sqrt(mean_squared_error(Y_train[800000:], RF.predict(X_train[800000:])))
            print a, b, c
            print 'Random forest RMSE - training set = %0.5f' % RF_rmse_train
            print 'Random forest RMSE - validation set = %0.5f' % RF_rmse_val

# random forest regressor - training & test split
RF = RandomForestRegressor(n_estimators=100, max_features=0.5, min_samples_leaf=1)
RF.fit(X_train, Y_train)
RF_pred = RF.predict(X_test)
RF_rmse = math.sqrt(mean_squared_error(Y_train, RF.predict(X_train)))
print 'New random forest RMSE = %0.5f' % RF_rmse
```

C.2 Generate Similarity Coefficients with RDKit

```
# generate similarity coefficients for all_smiles compared to a representative list (sample_fps)
def smiles_to_simil(all_smiles, sample_fps):
    mols = all_smiles.astype(str).apply(lambda x: Chem.MolFromSmiles(x))
    fps = mols.apply(lambda y: FingerprintMols.FingerprintMol(y)
    simils = fps.apply(lambda z: [round(fl, 5) for fl in DataStructs.BulkTanimotoSimilarity(z, sample_fps)])
    result_df = pd.DataFrame(np.vstack(simils))

    return result_df
```