# Practical 1 - WORKING DRAFT

Samuel Daulton, Taylor Killian, and Andrew Petschek (ML Marauders)

February 11, 2016

## 1 Technical Approach

### 1.1 Initial Attempts

Our initial approach to expand on the sample code provided to us for this machine learning problem was to implement Principal Component Analysis (PCA) to gain a better understanding of the interdependencies between features and to effectively reduce our feature set. We also performed some basic feature engineering upfront to couple with the PCA. There is a hypothesis that the number of branches influences the gap between HOMO and LUMO levels, and we pulled relevant compounds that were highlighted in a theoretical study on this topic conducted by Padtaya Pukpayat of the University of Nevada Las Vegas. This specific feature aggregates the number of active features, meaning if there are more features that are active in a compound, then there may be a higher orbital overlap and, thus, a lower HOMO-LUMO gap. A final initial feature we added was the presence of a benzene ring in the compound; we could identify a benzene ring which appeared to be more conjugated (i.e., closer together in energy). We combined this initial feature generation (referred to as Stage I Features in Results section below) with PCA, and turned to the existing methods to see if it we gained improvement over baseline. The results were promising, but we decided additional, more robust features needed to be engineered for improved impact.

### 1.2 Feature Engineering

To create new features, we turned to the RDKit. Specifically, we leveraged molecule object methods that returned integers. We felt integer outputs would work best with our modeling efforts, and we avoided methods that would output non-integers (e.g., strings and vectors). 114 new features (referred to as Stage II features in Results section below) were created in this way that spanned four main areas: general feature generation (e.g., number

of branches, number of bonds), descriptors (e.g., molecular weight, number of valence electrons), calculation of Lipinksi parameters, and fragment descriptors (e.g., number of aliphatic carboxylic acids). As stated above, these methods act on molecule objects, which take up a lot of memory, so our technical approach to feature generation was to divide the processing into four separate parts, which were ultimately merged together at the end.

In addition to using RDKit, we implemented our own algorithms to create additional features. We had a hypothesis that there likely existed interactions between features that would lead to larger HOMO-LUMO gaps. Thus, in addition to including a linear basis for each feature, we aimed to also include the product of pairs of features as well. Since we had well over 100 features, we decided we would look at the combinations between the top 25 most important features. To determine the top 25, we turned to Random Forests, which can output the importance weighting of each feature in our feature space. Using this regression output, we took the top 25 features and created $25C2$ (or 300) new features, which were the products of two features within an unique combinatoric pair (referred to as Stage III features in Results section below).

A final method for feature generation we used was based on logistic regression. Performing first some pre-processing, we rounded each gap value to the nearest half integer which we set as the labels we wished to predict. We ultimately used these labels as categorical features (referred to as Logistic in Results section below) that were included in our overall feature set.

### 1.3 Modeling Techniques

We not only built off of the simple linear regression and random forest model classes provided to us, but also approached this machine learning problem from other modeling angles as well. Across all methods, to prevent overfitting out-of-sample data sets when evaluating various hyperparameters for estimators, we split our training set into a training set and a validation set (67-33 split). This way, we avoided any knowledge of the test set leaking into our models and hurting their ability to generalize to new data. We had initially contemplated a cross-validation loop, where the creation of discrete training and validation sets would no longer be necessary to increase training sample size. However, since we had roughly one million samples to begin with, we decided we would have sufficient data to carry out our modeling using discrete training and validation sets. When estimating hyperparamters for each of our models below, we employed a grid search. Since we did not know which hyperparameter values may minimize our loss functions across models, we ran each model several times across an array of hyperparameter values ranging from small to large. At the end of the loop, we selected the hyperparameter value(s) that yielded a minimum loss when tested against our validation set and then re-trained the model on the

entire training set (including the validation set).

Examples of this grid search method were evident in three new methods we employed: Lasso Regression, Ridge Regression, and Elastic Net Regression (using library packages from scikit-learn). We felt attempting to use regularization techniques such as these would be a good idea as they would penalize very complex models and would likely fit out-of-sample data better. Lasso regression is a linear model with a regularization term (i.e., L1 prior). The constant $\alpha$ that multiplies the L1 term is a hyperparameter that can range between 0 and 1. We noticed that if $\alpha = 0$, then we have reduced this regression back to a simple linear least squares model without regularization. Ridge Regression is similar to Lasso, except it uses a L2 prior. Similarly, if $\alpha = 0$ in our Ridge Regression, we have reverted back to linear least squares. Elastic Net can be thought of as a combination of Lasso and Ridge Regression in that it combines both L1 and L2 priors as regularization terms. There are two hyperparameters in Elastic Net which measure the relationship between the L1 and L2 penalties, $\alpha$ and a ratio term between the two. Here, if $\alpha = 0$, we again have a linear least squares regression and if the ratio $= 0$ we have only Ridge Regression and if the ratio $= 1$ we have only Lasso Regression. Thus, we employed grid search to determine the right balance needed for loss minimization. With similar training-validation results across all three, we ultimately landed on using Lasso Regression for our final submission. Lasso regression is able to zero out features completely while Ridge Regression, for example, does not (given the differences between an absolute value function and a quadratic). For this reason, we decided Lasso Regression may be a better bet to generalize to the new test data set moving forward since we had hundreds of features.

In addition to linear modeling, we also employed ensemble methods (using library packages from scikit-learn), including Random Forests and Extra Trees Regressors. In our Random Forest regression we set the number of features per tree to be equal to $\frac{1}{\sqrt{p}}$ and used a grid search to compare number of estimators between 10 and 128. We had read in a paper by Oshiro et al. in 2012 that a good range of estimators is between 64-128, as anything more would result in additional computation time with minimal gains. We repeated this similar approach for Extra Trees Regressors. As stated above, Extra Trees is similar to Random Forests, where Extra Trees uses a random subset of features, but they differ from Random Forests in that thresholds are randomly determined for each feature where the best threshold is then picked for splitting. This additional set of randomness can lead to a reduction in variance and an increase in bias. For this reason, while we did run tests using Extra Trees Regressors, we ultimately decided to use Random Forests instead for their potentially lower bias and better ability to generalize to new data.

A few extensions were used across the methods above. The first involves PCA. We had initially thought that the use of PCA would be beneficial, as it would reveal the most important features in our feature space up front. We thought that this would improve performance. We found, however, that this did not seem to be the case. Reducing our

| Model | RMSE (Public Score) |
|---|---|
| Random Forest with Stage III Features + PCA | TBD |
| Random Forest with Stage III Features | 0.13842 |
| Lasso with Stage III Features | 0.16919 |
| Random Forest with Stage III Features + Logistic | 0.18256 |
| Lasso with Stage III Features + Logistic | 0.22629 |
| | |
| Extra Trees with Stage II Features + PCA | 0.15337 |
| Extra Trees with Stage II Features | 0.16863 |
| Lasso with Stage II Features | 0.22217 |
| | |
| Extra Trees with Stage I Features + PCA | 0.22972 |
| Extra Trees with Stage I Features | 0.22973 |
| Lasso with Stage I Features + PCA | 0.27116 |
| Lasso with Stage I Features | 0.29846 |
| | |
| Random Forest Baseline | 0.27207 |
| Linear Regression Baseline | 0.29846 |

Table 1: Model Results

feature space using PCA did not seem to improve our predictions (and in some cases it even led to negative impact). We also considered standardizing all non-binary features to ensure they are normally distributed. We felt normally distributed data would potentially lead to more accurate predictions. This too, like PCA, seemed to lead to worse results. Thus, we ultimately decided not to standardize our data. Finally, we tested models as we added new features along the way. This was a fluid process of adding features we felt to be better predictors; we did not have all features all at once at the start.

## 2 Results

We attempted many permutations of the above modeling techniques spanning the inclusion of various feature sets, models, hyperparameters, PCA, standardization etc... We found our best results to be when we used our full feature set and a Random Forrest Regressor with X trees. See table 1 for our RMSE results.

## 3 Discussion

This was an exciting opportunity to be able to work with a complex dataset on a machine learning problem with real-world implications. We approached this task as a prediction problem rather than an estimation problem. While prediction and estimation are similar, there is nuance between the two. While estimation concerns itself more with understanding the structural properties of the model and interpretation of coefficients, prediction cares most about loss minimization and a model's ability to generalize predictions to new data. Without context and heavy domain knowledge, it is hard to carry out estimation. We were however, able to carry out prediction.

A lot of our rationale for specific methods used in our modeling have been stated above in our Technical Approach section. Thus, they will not be repeated here. However, we will touch on several driving forces that, at a high level, dictated our actions throughout this project. As stated above in the technical approach section, we found feature engineering to be a critical component to our success. Our process involved adding a few features then running models, tuning parameters and making extensions until our RMSE values appeared to hit a wall. We would then look back at our features and brainstorm new features we could engineer. Once new features were added, we often saw an immediate bump in performance. We repeated this cycle several times throughout our time working on this project. We discovered that our modeling output could only be as good as the inputs we provide. Thus, better feature engineering provided our models with the regression inputs required for better training and prediction.

Another interesting nuance was PCA. We had initially believed PCA to be useful tool in prediction. We felt it would reduce our feature space to only the most predictive features. We found out however, that PCA did not help us much. But, before we realized this we would run a model twice: once with PCA and once without. This way we could better isolate the impact of PCA on our efforts. While PCA did not help much with our prediction accuracy, it did reduce computation time significantly. While computation time is not a benchmark that we are graded on, it is an integral component to scientific computing. As data sets become large, computation time required increases. Thus, one's ability to effectively reduce the time it takes to fit a model without sacrificing accuracy is highly valued. We often ran several models in parallel to speed up our analysis, but we could have considered additional parallel processing techniques (e.g., multi-processing, multi-threading) to speed up computation time.

Finally, with respect to our workflow, we carried out an array of models to get a handle on what seemed to work initially, yet we wondered if we could chain together modeling techniques to lead to better results. This rationale drove our final modeling efforts. We started with Random Forest regression to determine the best features in our data set. We then took those best features and used them to create 300 additional interaction features

which we then fed into a logistic classifier model. This logistic classifier model would provide additional classification features describing what each sample resembled in terms of HOMO-LUMO gap value. Once these features were made, we re-ran a new Random Forest model which could leverage these 300+ new features in addition to the hundreds that we already had.

Unfortunately, our logistic regression step used to classify data points did not translate to great results. While we had found very low training RMSE scores hovering around 0.11, they ultimately led to test set results of XXX. We feel this is due to an oversight in how the regression was performed. We took our entire training set and fit the logistic regression model onto it. We then used these predicted labels as part of our feature set used in the validation loop. In hindsight, we believe we should have used a seperate training set to fit the logistic classifier. Then take a new training set and predict labels using our logistic classifer. Then we could continue with our cross-validation procederues. Unfortunately, time and sample size were factors. Not only was there limited time to run our final Random Forest models, but we felt we may not have had a large enough sample size to create two effective training sets for the stated purposes above.

```
# coding: utf-8

# In[1]:

get_ipython().magic(u'matplotlib inline')
import pandas as pd
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Lasso, LassoCV, ElasticNetCV, SG
from sklearn.ensemble import RandomForestRegressor, ExtraTreesRegressor, AdaBoostRe
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV
from sklearn.cross_validation import train_test_split, KFold
from sklearn.metrics import mean_squared_error
from rdkit import DataStructs, Chem
from rdkit.Chem import AllChem, Descriptors, Lipinski, Fragments, rdmolops
from rdkit.ML.Cluster import Butina


# # *IF FEATURE ENGINEERING IS DONE: DO NOT START HERE*
# # *SKIP TO THE START OF SECTION IV: MODEL SELECTION*

# # I. Exploratory Data Analysis
```

```
# In[ ]:

# Read in training data
df_train = pd.read_csv("train.csv")


# In[ ]:

df_train.head()


# In[ ]:

plt.hist(df_train.gap.values, bins=50)


# ### Note:
# I see some outliers around -1.5.  Having negative values makes no sense, since th
==> Remove them

# In[ ]:

# Remove negative HOMO-LUMO gaps
df_train = df_train[df_train['gap'] >= 0]


# In[ ]:

print len(df_train)


# In[ ]:

# Plot histogram of gaps without negative values
plt.hist(df_train.gap.values, bins=50)


# # II. Training Data Processing and Feature Engineering
# 1. Drop all columns except smiles and gaps
# 2. Add "key" column, partition the data, and save those partitions
# 3. Remove all dataframes from memory
# 4. Load each partition (1 by 1), perform feature engineering, and save each parti
# 5. Reaggregate all partitions into one df, merge add binary-valued features from

# ### 1. Drop all columns except smiles and gaps
```

```
# In[ ]:

# remove all columns except smiles and gaps
df_train = df_train.drop(df_train.columns[range(1,257)], axis=1)
df_train.head()


# ### 2. Add "key" column, partition the data, and save those partitions

# In[ ]:

# add key column
df_train['key'] = df_train.index


# In[ ]:

df_train.head()


# In[ ]:

num_parts = 10
df_train_parts = np.array_split(df_train, num_parts)
for i in xrange(len(df_train_parts)):
    df_train_parts[i].to_csv('smiles_gaps_keys_df_train_part_'+str(i)+'.csv', index


# ### 3. Remove all dataframes from memory

# In[ ]:

del df_train
del df_train_parts


# ### 4. Load each partition (1 by 1), perform feature engineering, and save each p
#
# (http://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-f
#
# #### Number of Branches
# The idea is that the number of  [branches](https://en.wikipedia.org/wiki/Simplifi
#
# #### Note: We pull the specific compounds of intereste from [this poster.](http:/
#
# #### Benzene Ring
```

```
# The last feature is determining, from the SMILES encoding whether or not there is
#
# #### Number of Double Bonds
#
# #### Additional Features engineered using from RDKit library

# In[ ]:

for i in xrange(num_parts):
    df_train_part = pd.read_csv('smiles_gaps_keys_df_train_part_'+str(i)+'.csv')

    # Apply feature engineering to train data
    num_branches = df_train_part.smiles.apply(lambda x: x.count('('))
    has_benzothiophene = df_train_part.smiles.apply(lambda x: min(1,x.count('s2c1cc
    has_carbazole = df_train_part.smiles.apply(lambda x: min(1,x.count('c1ccc2c(c1)
    has_fluorene = df_train_part.smiles.apply(lambda x: min(1,x.count('c1ccc-2c(c1)
    num_double_bonds = df_train_part.smiles.apply(lambda x: x.count('='))
    df_train_part['num_branches'] = num_branches
    df_train_part['has_benzothiophene'] = has_benzothiophene
    df_train_part['has_carbazole'] = has_carbazole
    df_train_part['has_fluorene'] = has_fluorene
    df_train_part['num_double_bonds'] = num_double_bonds
    print "Part {}: finished initial feat gen".format(i)
    # RDKit Feature Engineering
    # Generate molecule objects
    molecules = df_train_part.smiles.apply(lambda x: Chem.MolFromSmiles(x))
    print "Part {}: finished molecule generation".format(i)
    # Generate Features
    df_train_part['avg_molecular_weight'] = molecules.apply(lambda x: Descriptors.M
    print "Part {}: finished first feature".format(i)
    df_train_part['exact_molecular_weight'] = molecules.apply(lambda x: Descriptors
    df_train_part['avg_molecular_weight_ignore_hydrogen'] = molecules.apply(lambda
    df_train_part['num_valence_electrons'] = molecules.apply(lambda x: Descriptors.
    df_train_part['num_radical_electrons'] = molecules.apply(lambda x: Descriptors.
    df_train_part['formal_charge'] = molecules.apply(lambda x: rdmolops.GetFormalCh
    df_train_part['sssr'] = molecules.apply(lambda x: rdmolops.GetSSSR(x))
    print "Part {}: finished Descriptors and rdmolops".format(i)
    df_train_part['fraction_csp3'] = molecules.apply(lambda x: Lipinski.FractionCSP
    df_train_part['num_aliphatic_carbocycles'] = molecules.apply(lambda x: Lipinski
    df_train_part['num_aliphatic_heterocycles'] = molecules.apply(lambda x: Lipinsk
    df_train_part['num_aliphatic_rings'] = molecules.apply(lambda x: Lipinski.NumAl
    df_train_part['num_aromatic_carbocycles'] = molecules.apply(lambda x: Lipinski.
    df_train_part['num_aromatic_heterocycles'] = molecules.apply(lambda x: Lipinski
    df_train_part['num_aromatic_rings'] = molecules.apply(lambda x: Lipinski.NumAro
    df_train_part['num_saturated_carbocycles'] = molecules.apply(lambda x: Lipinski
    df_train_part['num_saturated_heterocycles'] = molecules.apply(lambda x: Lipinsk
```

```
df_train_part['num_saturated_rings'] = molecules.apply(lambda x: Lipinski.NumSa
df_train_part['num_nh_oh'] = molecules.apply(lambda x: Lipinski.NHOHCount(x))
df_train_part['num_num_rotatable_bonds'] = molecules.apply(lambda x: Lipinski.N
df_train_part['num_heteroatoms'] = molecules.apply(lambda x: Lipinski.NumHetero
df_train_part['num_h_acceptors'] = molecules.apply(lambda x: Lipinski.NumHAccep
df_train_part['num_h_donors'] = molecules.apply(lambda x: Lipinski.NumHDonors(x
df_train_part['ring_count'] = molecules.apply(lambda x: Lipinski.RingCount(x))
print "Part {}: finished Lipinski".format(i)
# See Parsing_methods_from_rdk_source.ipynb
df_train_part['fr_Al_COO'] = molecules.apply(lambda x: Fragments.fr_Al_COO(x))
df_train_part['fr_Al_OH'] = molecules.apply(lambda x: Fragments.fr_Al_OH(x))
df_train_part['fr_Al_OH_noTert'] = molecules.apply(lambda x: Fragments.fr_Al_OH
df_train_part['fr_ArN'] = molecules.apply(lambda x: Fragments.fr_ArN(x))
df_train_part['fr_Ar_COO'] = molecules.apply(lambda x: Fragments.fr_Ar_COO(x))
df_train_part['fr_Ar_N'] = molecules.apply(lambda x: Fragments.fr_Ar_N(x))
df_train_part['fr_Ar_NH'] = molecules.apply(lambda x: Fragments.fr_Ar_NH(x))
df_train_part['fr_Ar_OH'] = molecules.apply(lambda x: Fragments.fr_Ar_OH(x))
df_train_part['fr_COO'] = molecules.apply(lambda x: Fragments.fr_COO(x))
df_train_part['fr_COO2'] = molecules.apply(lambda x: Fragments.fr_COO2(x))
df_train_part['fr_C_O'] = molecules.apply(lambda x: Fragments.fr_C_O(x))
df_train_part['fr_C_O_noCOO'] = molecules.apply(lambda x: Fragments.fr_C_O_noCO
df_train_part['fr_C_S'] = molecules.apply(lambda x: Fragments.fr_C_S(x))
df_train_part['fr_HOCCN'] = molecules.apply(lambda x: Fragments.fr_HOCCN(x))
df_train_part['fr_Imine'] = molecules.apply(lambda x: Fragments.fr_Imine(x))
df_train_part['fr_NH0'] = molecules.apply(lambda x: Fragments.fr_NH0(x))
df_train_part['fr_NH1'] = molecules.apply(lambda x: Fragments.fr_NH1(x))
df_train_part['fr_NH2'] = molecules.apply(lambda x: Fragments.fr_NH2(x))
df_train_part['fr_N_O'] = molecules.apply(lambda x: Fragments.fr_N_O(x))
df_train_part['fr_Ndealkylation1'] = molecules.apply(lambda x: Fragments.fr_Nde
df_train_part['fr_Ndealkylation2'] = molecules.apply(lambda x: Fragments.fr_Nde
df_train_part['fr_Nhpyrrole'] = molecules.apply(lambda x: Fragments.fr_Nhpyrrol
df_train_part['fr_SH'] = molecules.apply(lambda x: Fragments.fr_SH(x))
df_train_part['fr_aldehyde'] = molecules.apply(lambda x: Fragments.fr_aldehyde(
df_train_part['fr_alkyl_carbamate'] = molecules.apply(lambda x: Fragments.fr_al
df_train_part['fr_alkyl_halide'] = molecules.apply(lambda x: Fragments.fr_alkyl
df_train_part['fr_allylic_oxid'] = molecules.apply(lambda x: Fragments.fr_allyl
df_train_part['fr_amide'] = molecules.apply(lambda x: Fragments.fr_amide(x))
df_train_part['fr_amidine'] = molecules.apply(lambda x: Fragments.fr_amidine(x)
df_train_part['fr_aniline'] = molecules.apply(lambda x: Fragments.fr_aniline(x)
df_train_part['fr_aryl_methyl'] = molecules.apply(lambda x: Fragments.fr_aryl_m
df_train_part['fr_azide'] = molecules.apply(lambda x: Fragments.fr_azide(x))
df_train_part['fr_azo'] = molecules.apply(lambda x: Fragments.fr_azo(x))
df_train_part['fr_barbitur'] = molecules.apply(lambda x: Fragments.fr_barbitur(
df_train_part['fr_benzene'] = molecules.apply(lambda x: Fragments.fr_benzene(x)
df_train_part['fr_benzodiazepine'] = molecules.apply(lambda x: Fragments.fr_ben
df_train_part['fr_bicyclic'] = molecules.apply(lambda x: Fragments.fr_bicyclic(
```

```
df_train_part['fr_diazo'] = molecules.apply(lambda x: Fragments.fr_diazo(x))
df_train_part['fr_dihydropyridine'] = molecules.apply(lambda x: Fragments.fr_di
df_train_part['fr_epoxide'] = molecules.apply(lambda x: Fragments.fr_epoxide(x)
df_train_part['fr_ester'] = molecules.apply(lambda x: Fragments.fr_ester(x))
df_train_part['fr_ether'] = molecules.apply(lambda x: Fragments.fr_ether(x))
df_train_part['fr_furan'] = molecules.apply(lambda x: Fragments.fr_furan(x))
df_train_part['fr_guanido'] = molecules.apply(lambda x: Fragments.fr_guanido(x)
df_train_part['fr_halogen'] = molecules.apply(lambda x: Fragments.fr_halogen(x)
df_train_part['fr_hdrzine'] = molecules.apply(lambda x: Fragments.fr_hdrzine(x)
df_train_part['fr_hdrzone'] = molecules.apply(lambda x: Fragments.fr_hdrzone(x)
df_train_part['fr_imidazole'] = molecules.apply(lambda x: Fragments.fr_imidazol
df_train_part['fr_imide'] = molecules.apply(lambda x: Fragments.fr_imide(x))
df_train_part['fr_isocyan'] = molecules.apply(lambda x: Fragments.fr_isocyan(x)
df_train_part['fr_isothiocyan'] = molecules.apply(lambda x: Fragments.fr_isothi
df_train_part['fr_ketone'] = molecules.apply(lambda x: Fragments.fr_ketone(x))
df_train_part['fr_lactam'] = molecules.apply(lambda x: Fragments.fr_lactam(x))
df_train_part['fr_lactone'] = molecules.apply(lambda x: Fragments.fr_lactone(x)
df_train_part['fr_methoxy'] = molecules.apply(lambda x: Fragments.fr_methoxy(x)
df_train_part['fr_morpholine'] = molecules.apply(lambda x: Fragments.fr_morphol
df_train_part['fr_nitrile'] = molecules.apply(lambda x: Fragments.fr_nitrile(x)
df_train_part['fr_nitro'] = molecules.apply(lambda x: Fragments.fr_nitro(x))
df_train_part['fr_nitro_arom'] = molecules.apply(lambda x: Fragments.fr_nitro_a
df_train_part['fr_nitro_arom_nonortho'] = molecules.apply(lambda x: Fragments.f
df_train_part['fr_nitroso'] = molecules.apply(lambda x: Fragments.fr_nitroso(x)
df_train_part['fr_oxazole'] = molecules.apply(lambda x: Fragments.fr_oxazole(x)
df_train_part['fr_oxime'] = molecules.apply(lambda x: Fragments.fr_oxime(x))
df_train_part['fr_para_hydroxylation'] = molecules.apply(lambda x: Fragments.fr
df_train_part['fr_phenol'] = molecules.apply(lambda x: Fragments.fr_phenol(x))
df_train_part['fr_phenol_noOrthoHbond'] = molecules.apply(lambda x: Fragments.f
df_train_part['fr_phos_acid'] = molecules.apply(lambda x: Fragments.fr_phos_aci
df_train_part['fr_phos_ester'] = molecules.apply(lambda x: Fragments.fr_phos_es
df_train_part['fr_piperdine'] = molecules.apply(lambda x: Fragments.fr_piperdin
df_train_part['fr_piperzine'] = molecules.apply(lambda x: Fragments.fr_piperzin
df_train_part['fr_priamide'] = molecules.apply(lambda x: Fragments.fr_priamide(
df_train_part['fr_prisulfonamd'] = molecules.apply(lambda x: Fragments.fr_prisu
df_train_part['fr_pyridine'] = molecules.apply(lambda x: Fragments.fr_pyridine(
df_train_part['fr_quatN'] = molecules.apply(lambda x: Fragments.fr_quatN(x))
df_train_part['fr_sulfide'] = molecules.apply(lambda x: Fragments.fr_sulfide(x)
df_train_part['fr_sulfonamd'] = molecules.apply(lambda x: Fragments.fr_sulfonam
df_train_part['fr_sulfone'] = molecules.apply(lambda x: Fragments.fr_sulfone(x)
df_train_part['fr_term_acetylene'] = molecules.apply(lambda x: Fragments.fr_ter
df_train_part['fr_tetrazole'] = molecules.apply(lambda x: Fragments.fr_tetrazol
df_train_part['fr_thiazole'] = molecules.apply(lambda x: Fragments.fr_thiazole(
df_train_part['fr_thiocyan'] = molecules.apply(lambda x: Fragments.fr_thiocyan(
df_train_part['fr_thiophene'] = molecules.apply(lambda x: Fragments.fr_thiophen
df_train_part['fr_unbrch_alkane'] = molecules.apply(lambda x: Fragments.fr_unbr
```

```
    df_train_part['fr_urea'] = molecules.apply(lambda x: Fragments.fr_urea(x))
    df_train_part['fr_ketone_Topliss'] = molecules.apply(lambda x: Fragments.fr_ket

    print "Part {}: finished Fragments".format(i)
    df_train_part.to_csv('rdk_feat_eng_df_train_part_'+str(i)+'.csv', index=False)
    del df_train_part
    del molecules


# ### 5. Reaggregate all partitions into one df, merge add binary-valued features f

# In[ ]:

# Read in each feature engineered partition
df_train_parts = [pd.read_csv('rdk_feat_eng_df_train_part_'+str(i)+'.csv') for i in
# concatenate them into one df
df_train = pd.concat(df_train_parts)


# In[ ]:

# Read in old training
# Note: df_train_old has same as before partitioning
df_train_old = pd.read_csv("train.csv")
# Add key
df_train_old['key'] = df_train_old.index
# merge dataframes on 'key'
df_train = df_train.merge(df_train_old, on=["key"])
# Fix columns
df_train = df_train.drop(['gap_x'],axis=1)
df_train = df_train.drop(['key'],axis=1)
df_train = df_train.rename(columns = {'gap_y':'gap'})
df_train = df_train.drop(['smiles_x'],axis=1)
df_train = df_train.rename(columns = {'smiles_y':'smiles'})


# In[ ]:

df_train.head()


# In[ ]:

# Save df_train
df_train.to_csv('FINAL_train.csv', index=False)
```

```
# In[ ]:

# Remove df_train for now
del df_train_parts
del df_train_old
del df_train


# # III. Test Data Processing and Feature Engineering
# 1. Drop all columns except smiles and ids
# 2. Partition the data and save those partitions
# 3. Remove all dataframes from memory
# 4. Load each partition (1 by 1), perform feature engineering, and save each parti
# 5. Reaggregate all partitions into one df, merge add binary-valued features from

# In[ ]:

# Read in test data
df_test = pd.read_csv("test.csv")
df_test.head()


# ### 1. Drop all columns except smiles and ids

# In[ ]:

df_test = df_test.drop(df_test.columns[range(2,258)], axis=1)


# In[ ]:

df_test.head()


# ### 2. Partition the data and save those partitions

# In[ ]:

num_parts = 10


# In[ ]:

df_test_parts = np.array_split(df_test, num_parts)
for i in xrange(num_parts):
```

```
    df_test_parts[i].to_csv('smiles_and_ids_df_test_part_'+str(i)+'.csv', index=Fal


# ### 3. Remove all dataframes from memory

# In[ ]:

del df_test
del df_test_parts


# ### 4. Load each partition (1 by 1), perform feature engineering, and save each p

# In[ ]:

for i in xrange(num_parts):
    df_test_part = pd.read_csv('smiles_and_ids_df_test_part_'+str(i)+'.csv')


    # Apply feature engineering to train data
    num_branches = df_test_part.smiles.apply(lambda x: x.count('('))
    has_benzothiophene = df_test_part.smiles.apply(lambda x: min(1,x.count('s2c1ccc
    has_carbazole = df_test_part.smiles.apply(lambda x: min(1,x.count('c1ccc2c(c1)c
    has_fluorene = df_test_part.smiles.apply(lambda x: min(1,x.count('c1ccc-2c(c1)C
    num_double_bonds = df_test_part.smiles.apply(lambda x: x.count('='))
    df_test_part['num_branches'] = num_branches
    df_test_part['has_benzothiophene'] = has_benzothiophene
    df_test_part['has_carbazole'] = has_carbazole
    df_test_part['has_fluorene'] = has_fluorene
    df_test_part['num_double_bonds'] = num_double_bonds
    print "Part {}: finished initial feat gen".format(i)
    # RDKit Feature Engineering
    # Generate molecule objects
    molecules = df_test_part.smiles.apply(lambda x: Chem.MolFromSmiles(x))
    print "Part {}: finished molecule generation".format(i)
    # Generate Features
    df_test_part['avg_molecular_weight'] = molecules.apply(lambda x: Descriptors.Mo
    print "Part {}: finished first feature".format(i)
    df_test_part['exact_molecular_weight'] = molecules.apply(lambda x: Descriptors.
    df_test_part['avg_molecular_weight_ignore_hydrogen'] = molecules.apply(lambda x
    df_test_part['num_valence_electrons'] = molecules.apply(lambda x: Descriptors.N
    df_test_part['num_radical_electrons'] = molecules.apply(lambda x: Descriptors.N
    df_test_part['formal_charge'] = molecules.apply(lambda x: rdmolops.GetFormalCha
    df_test_part['sssr'] = molecules.apply(lambda x: rdmolops.GetSSSR(x))
    print "Part {}: finished Descriptors and rdmolops".format(i)
    df_test_part['fraction_csp3'] = molecules.apply(lambda x: Lipinski.FractionCSP3
```

```
df_test_part['num_aliphatic_carbocycles'] = molecules.apply(lambda x: Lipinski.
df_test_part['num_aliphatic_heterocycles'] = molecules.apply(lambda x: Lipinski
df_test_part['num_aliphatic_rings'] = molecules.apply(lambda x: Lipinski.NumAli
df_test_part['num_aromatic_carbocycles'] = molecules.apply(lambda x: Lipinski.N
df_test_part['num_aromatic_heterocycles'] = molecules.apply(lambda x: Lipinski.
df_test_part['num_aromatic_rings'] = molecules.apply(lambda x: Lipinski.NumArom
df_test_part['num_saturated_carbocycles'] = molecules.apply(lambda x: Lipinski.
df_test_part['num_saturated_heterocycles'] = molecules.apply(lambda x: Lipinski
df_test_part['num_saturated_rings'] = molecules.apply(lambda x: Lipinski.NumSat
df_test_part['num_nh_oh'] = molecules.apply(lambda x: Lipinski.NHOHCount(x))
df_test_part['num_num_rotatable_bonds'] = molecules.apply(lambda x: Lipinski.Nu
df_test_part['num_heteroatoms'] = molecules.apply(lambda x: Lipinski.NumHeteroa
df_test_part['num_h_acceptors'] = molecules.apply(lambda x: Lipinski.NumHAccept
df_test_part['num_h_donors'] = molecules.apply(lambda x: Lipinski.NumHDonors(x)
df_test_part['ring_count'] = molecules.apply(lambda x: Lipinski.RingCount(x))
print "Part {}: finished Lipinski".format(i)
# See Parsing_methods_from_rdk_source.ipynb
df_test_part['fr_Al_COO'] = molecules.apply(lambda x: Fragments.fr_Al_COO(x))
df_test_part['fr_Al_OH'] = molecules.apply(lambda x: Fragments.fr_Al_OH(x))
df_test_part['fr_Al_OH_noTert'] = molecules.apply(lambda x: Fragments.fr_Al_OH_
df_test_part['fr_ArN'] = molecules.apply(lambda x: Fragments.fr_ArN(x))
df_test_part['fr_Ar_COO'] = molecules.apply(lambda x: Fragments.fr_Ar_COO(x))
df_test_part['fr_Ar_N'] = molecules.apply(lambda x: Fragments.fr_Ar_N(x))
df_test_part['fr_Ar_NH'] = molecules.apply(lambda x: Fragments.fr_Ar_NH(x))
df_test_part['fr_Ar_OH'] = molecules.apply(lambda x: Fragments.fr_Ar_OH(x))
df_test_part['fr_COO'] = molecules.apply(lambda x: Fragments.fr_COO(x))
df_test_part['fr_COO2'] = molecules.apply(lambda x: Fragments.fr_COO2(x))
df_test_part['fr_C_O'] = molecules.apply(lambda x: Fragments.fr_C_O(x))
df_test_part['fr_C_O_noCOO'] = molecules.apply(lambda x: Fragments.fr_C_O_noCOO
df_test_part['fr_C_S'] = molecules.apply(lambda x: Fragments.fr_C_S(x))
df_test_part['fr_HOCCN'] = molecules.apply(lambda x: Fragments.fr_HOCCN(x))
df_test_part['fr_Imine'] = molecules.apply(lambda x: Fragments.fr_Imine(x))
df_test_part['fr_NH0'] = molecules.apply(lambda x: Fragments.fr_NH0(x))
df_test_part['fr_NH1'] = molecules.apply(lambda x: Fragments.fr_NH1(x))
df_test_part['fr_NH2'] = molecules.apply(lambda x: Fragments.fr_NH2(x))
df_test_part['fr_N_O'] = molecules.apply(lambda x: Fragments.fr_N_O(x))
df_test_part['fr_Ndealkylation1'] = molecules.apply(lambda x: Fragments.fr_Ndea
df_test_part['fr_Ndealkylation2'] = molecules.apply(lambda x: Fragments.fr_Ndea
df_test_part['fr_Nhpyrrole'] = molecules.apply(lambda x: Fragments.fr_Nhpyrrole
df_test_part['fr_SH'] = molecules.apply(lambda x: Fragments.fr_SH(x))
df_test_part['fr_aldehyde'] = molecules.apply(lambda x: Fragments.fr_aldehyde(x
df_test_part['fr_alkyl_carbamate'] = molecules.apply(lambda x: Fragments.fr_alk
df_test_part['fr_alkyl_halide'] = molecules.apply(lambda x: Fragments.fr_alkyl_
df_test_part['fr_allylic_oxid'] = molecules.apply(lambda x: Fragments.fr_allyli
df_test_part['fr_amide'] = molecules.apply(lambda x: Fragments.fr_amide(x))
df_test_part['fr_amidine'] = molecules.apply(lambda x: Fragments.fr_amidine(x))
```

```
df_test_part['fr_aniline'] = molecules.apply(lambda x: Fragments.fr_aniline(x))
df_test_part['fr_aryl_methyl'] = molecules.apply(lambda x: Fragments.fr_aryl_me
df_test_part['fr_azide'] = molecules.apply(lambda x: Fragments.fr_azide(x))
df_test_part['fr_azo'] = molecules.apply(lambda x: Fragments.fr_azo(x))
df_test_part['fr_barbitur'] = molecules.apply(lambda x: Fragments.fr_barbitur(x
df_test_part['fr_benzene'] = molecules.apply(lambda x: Fragments.fr_benzene(x))
df_test_part['fr_benzodiazepine'] = molecules.apply(lambda x: Fragments.fr_benz
df_test_part['fr_bicyclic'] = molecules.apply(lambda x: Fragments.fr_bicyclic(x
df_test_part['fr_diazo'] = molecules.apply(lambda x: Fragments.fr_diazo(x))
df_test_part['fr_dihydropyridine'] = molecules.apply(lambda x: Fragments.fr_dih
df_test_part['fr_epoxide'] = molecules.apply(lambda x: Fragments.fr_epoxide(x))
df_test_part['fr_ester'] = molecules.apply(lambda x: Fragments.fr_ester(x))
df_test_part['fr_ether'] = molecules.apply(lambda x: Fragments.fr_ether(x))
df_test_part['fr_furan'] = molecules.apply(lambda x: Fragments.fr_furan(x))
df_test_part['fr_guanido'] = molecules.apply(lambda x: Fragments.fr_guanido(x))
df_test_part['fr_halogen'] = molecules.apply(lambda x: Fragments.fr_halogen(x))
df_test_part['fr_hdrzine'] = molecules.apply(lambda x: Fragments.fr_hdrzine(x))
df_test_part['fr_hdrzone'] = molecules.apply(lambda x: Fragments.fr_hdrzone(x))
df_test_part['fr_imidazole'] = molecules.apply(lambda x: Fragments.fr_imidazole
df_test_part['fr_imide'] = molecules.apply(lambda x: Fragments.fr_imide(x))
df_test_part['fr_isocyan'] = molecules.apply(lambda x: Fragments.fr_isocyan(x))
df_test_part['fr_isothiocyan'] = molecules.apply(lambda x: Fragments.fr_isothio
df_test_part['fr_ketone'] = molecules.apply(lambda x: Fragments.fr_ketone(x))
df_test_part['fr_lactam'] = molecules.apply(lambda x: Fragments.fr_lactam(x))
df_test_part['fr_lactone'] = molecules.apply(lambda x: Fragments.fr_lactone(x))
df_test_part['fr_methoxy'] = molecules.apply(lambda x: Fragments.fr_methoxy(x))
df_test_part['fr_morpholine'] = molecules.apply(lambda x: Fragments.fr_morpholi
df_test_part['fr_nitrile'] = molecules.apply(lambda x: Fragments.fr_nitrile(x))
df_test_part['fr_nitro'] = molecules.apply(lambda x: Fragments.fr_nitro(x))
df_test_part['fr_nitro_arom'] = molecules.apply(lambda x: Fragments.fr_nitro_ar
df_test_part['fr_nitro_arom_nonortho'] = molecules.apply(lambda x: Fragments.fr
df_test_part['fr_nitroso'] = molecules.apply(lambda x: Fragments.fr_nitroso(x))
df_test_part['fr_oxazole'] = molecules.apply(lambda x: Fragments.fr_oxazole(x))
df_test_part['fr_oxime'] = molecules.apply(lambda x: Fragments.fr_oxime(x))
df_test_part['fr_para_hydroxylation'] = molecules.apply(lambda x: Fragments.fr_
df_test_part['fr_phenol'] = molecules.apply(lambda x: Fragments.fr_phenol(x))
df_test_part['fr_phenol_noOrthoHbond'] = molecules.apply(lambda x: Fragments.fr
df_test_part['fr_phos_acid'] = molecules.apply(lambda x: Fragments.fr_phos_acid
df_test_part['fr_phos_ester'] = molecules.apply(lambda x: Fragments.fr_phos_est
df_test_part['fr_piperdine'] = molecules.apply(lambda x: Fragments.fr_piperdine
df_test_part['fr_piperzine'] = molecules.apply(lambda x: Fragments.fr_piperzine
df_test_part['fr_priamide'] = molecules.apply(lambda x: Fragments.fr_priamide(x
df_test_part['fr_prisulfonamd'] = molecules.apply(lambda x: Fragments.fr_prisul
df_test_part['fr_pyridine'] = molecules.apply(lambda x: Fragments.fr_pyridine(x
df_test_part['fr_quatN'] = molecules.apply(lambda x: Fragments.fr_quatN(x))
df_test_part['fr_sulfide'] = molecules.apply(lambda x: Fragments.fr_sulfide(x))
```

```
    df_test_part['fr_sulfonamd'] = molecules.apply(lambda x: Fragments.fr_sulfonamd
    df_test_part['fr_sulfone'] = molecules.apply(lambda x: Fragments.fr_sulfone(x))
    df_test_part['fr_term_acetylene'] = molecules.apply(lambda x: Fragments.fr_term
    df_test_part['fr_tetrazole'] = molecules.apply(lambda x: Fragments.fr_tetrazole
    df_test_part['fr_thiazole'] = molecules.apply(lambda x: Fragments.fr_thiazole(x
    df_test_part['fr_thiocyan'] = molecules.apply(lambda x: Fragments.fr_thiocyan(x
    df_test_part['fr_thiophene'] = molecules.apply(lambda x: Fragments.fr_thiophene
    df_test_part['fr_unbrch_alkane'] = molecules.apply(lambda x: Fragments.fr_unbrc
    df_test_part['fr_urea'] = molecules.apply(lambda x: Fragments.fr_urea(x))
    df_test_part['fr_ketone_Topliss'] = molecules.apply(lambda x: Fragments.fr_keto

    print "Part {}: finished Fragments".format(i)
    df_test_part.to_csv('rdk_feat_eng_df_test_part_'+str(i)+'.csv', index=False)
    del df_test_part
    del molecules


# ### 5. Reaggregate all partitions into one df, merge add binary-valued features f

# In[ ]:

# read in each feat eng part
df_test_parts = [pd.read_csv('rdk_feat_eng_df_test_part_'+str(i)+'.csv') for i in x


# In[ ]:

# combine into one df
df_test = pd.concat(df_test_parts)


# In[ ]:

# Read in old test data and merge with new features
df_test_old = pd.read_csv("test.csv")
df_test = df_test.merge(df_test_old, on=["Id"])
# Fix columns
df_test = df_test.drop(['smiles_x'],axis=1)
df_test = df_test.rename(columns = {'smiles_y':'smiles'})
df_test.head()


# In[ ]:

# Save to csv
df_test.to_csv('FINAL_test.csv', index=False)
```

```
# In[ ]:

# Delete old dfs
del df_test_old
del df_test_parts
del df_test


# # *IF FEATURE ENGINEERING IS DONE: START HERE*
# # IV. Model Selection
# 1. Read training data, store output values, and remove output values from df_trai
# 2. Split training data into training set and validation set
# 3. Model Selection -  Test various models

# ### 1. Read training data, store output values, and remove output values from df_

# In[2]:

df_train = pd.read_csv('FINAL_train.csv')


# In[3]:

df_train.head()


# In[4]:

# Drop the 'smiles' column
smiles = df_train.smiles.values
df_train = df_train.drop(['smiles'], axis=1)

# Store gap values
Y_train = df_train.gap.values

# Delete 'gap' column
df_train = df_train.drop(['gap'], axis=1)
X_train = df_train.values
print "Train features:", X_train.shape
print "Train gap:", Y_train.shape


# In[8]:
```

```
# train rf regressor
rf_est = RandomForestRegressor(n_estimators=128,n_jobs=2, max_features='sqrt')
rf_est.fit(X_train, Y_train)


# ## Feature Selection

# In[9]:

# get feature names and importances
feats = zip(df_train.columns, rf_est.feature_importances_)
feats = sorted(feats, key=lambda x: x[1], reverse=True)
feats


# In[10]:

# select top 25 ranked features
top_feats = zip(*feats[:25])[0]
top_feats


# In[11]:

# create interaction terms (e.g. x_1 * x_2) for all top features
for i in range(len(top_feats)):
    for j in range(i+1,len(top_feats)):
        feat_i = top_feats[i]
        feat_j = top_feats[j]
        df_train[feat_i+'-'+feat_j] = df_train[feat_i]*df_train[feat_j]


# In[12]:

df_train['smiles'] = smiles
df_train['gap'] = Y_train


# In[13]:

df_train.to_csv('FINAL_train_25_interactions.csv', index=False)


# In[14]:

del df_train
```

```
# ####  Add features for test data

# In[15]:

df_test = pd.read_csv("FINAL_test.csv")


# In[17]:

# create interaction terms (e.g. x_1 * x_2) for all top features
for i in range(len(top_feats)):
    for j in range(i+1,len(top_feats)):
        feat_i = top_feats[i]
        feat_j = top_feats[j]
        df_test[feat_i+'-'+feat_j] = df_test[feat_i]*df_test[feat_j]


# In[18]:

df_test.head()


# In[19]:

df_test.to_csv('FINAL_test_25_interactions.csv', index=False)


# In[20]:

del df_test


# ### 2. Split training data into training set and validation set

# In[21]:

df_train = pd.read_csv('FINAL_train_25_interactions.csv')


# In[22]:

# Drop the 'smiles' column
df_train = df_train.drop(['smiles'], axis=1)
```

```
# Store gap values
Y_train = df_train.gap.values

# Delete 'gap' column
df_train = df_train.drop(['gap'], axis=1)
X_train = df_train.values
print "Train features:", X_train.shape
print "Train gap:", Y_train.shape


# In[23]:

# Partition Training Data into Training, Validation
cross_X_train, cross_X_valid, cross_Y_train, cross_Y_valid = train_test_split(X_tra


# ### 3. Model Selection -  Test various models
# #### A. Linear Regression

# In[24]:

# Fit Linear Regression to cross_X_train and validate it on validations set
LR = LinearRegression()
LR.fit(cross_X_train, cross_Y_train)
LR_pred = LR.predict(cross_X_valid)


# In[25]:

LR.coef_
zero_coefs = []
for i in xrange(len(LR.coef_)):
    if LR.coef_[i] == 0:
        zero_coefs.append(i)
print "Number of coefficients that are zero:", len(zero_coefs)
print "Total number of coefficients:", len(LR.coef_)
print zero_coefs


# In[26]:

print LR_pred


# In[27]:
```

```
mean_squared_error(cross_Y_valid, LR_pred)**.5


# #### B. Transform data using PCA

# In[ ]:

pca = PCA(n_components=60)
cross_X_train_transf = pca.fit_transform(cross_X_train)
cross_X_valid_transf = pca.transform(cross_X_valid)


# #### No PCA, Random Forest, feature selection

# In[28]:

# train rf regressor
rf_est = RandomForestRegressor(n_estimators=128,n_jobs=2, max_features='sqrt')
rf_est.fit(cross_X_train, cross_Y_train)


# In[29]:

rf_pred = rf_est.predict(cross_X_valid)


# In[30]:

#RMSE
mean_squared_error(cross_Y_valid, rf_pred)**.5


# #### C. Random Forest (extra trees) using PCA

# In[ ]:

extraTrees_pca = ExtraTreesRegressor(n_estimators=100,n_jobs=2)
tree_est_wPCA = extraTrees_pca.fit(cross_X_train_transf, cross_Y_train)
pca_exTree_pred = tree_est_wPCA.predict(cross_X_valid_transf)


# In[ ]:

# Calculate RMSE
mean_squared_error(cross_Y_valid, pca_exTree_pred)**.5
```

```
# #### D1. Lasso

# In[ ]:

from sklearn import linear_model
from sklearn.metrics import mean_squared_error

# Set parameters to test
# alphas = np.logspace(-4, -.5, 30) alpha = 0.0001 was best with RMSE of 0.25
alphas = [.5,.1,.01,.001,.0001,.00001]

# Initialize minimums

minimum_alpha = 100
minimum_RMSE = 100
for alpha in alphas:

    # Fit model and predict on validation
    clf = linear_model.Lasso(alpha=alpha)
    clf.fit(cross_X_train,cross_Y_train)
    y_pred = clf.predict(cross_X_valid)

    # Calculate RMSE and update minimum RMSE if possible
    RMSE = np.sqrt(mean_squared_error(cross_Y_valid, y_pred))
    if RMSE < minimum_RMSE:
        minimum_RMSE = RMSE
        minimum_alpha = alpha

print "minimum RMSE is", minimum_RMSE
print "minimum alpha is",minimum_alpha


# #### D2. Lasso using PCA

# In[ ]:

from sklearn import linear_model
from sklearn.metrics import mean_squared_error

# Set parameters to test
# alphas = np.logspace(-4, -.5, 30) alpha = 0.0001 was best with RMSE of 0.25
alphas = [.5,.1,.01,.001,.0001,.00001]

# Initialize minimums
```

```
minimum_alpha = 100
minimum_RMSE = 100
for alpha in alphas:

    # Fit model and predict on validation
    clf = linear_model.Lasso(alpha=alpha)
    clf.fit(cross_X_train_transf,cross_Y_train)
    y_pred = clf.predict(cross_X_valid_transf)

    # Calculate RMSE and update minimum RMSE if possible
    RMSE = np.sqrt(mean_squared_error(cross_Y_valid, y_pred))
    if RMSE < minimum_RMSE:
        minimum_RMSE = RMSE
        minimum_alpha = alpha

print "minimum RMSE is", minimum_RMSE
print "minimum alpha is",minimum_alpha


# #### D1. Elastic Net

# In[ ]:

# Set parameters to test
alphas = [.5,.1,.01,.001,.0001,.00001]
ratios = [.5,.1,.01,.001,.0001,.00001]
counter = 0

# Initialize minimums
minimum_alpha = 100
minimum_ratio = 100
minimum_RMSE = 100
for alpha in alphas:
    for ratio in ratios:

        # Fit model and predict on validation
        clf = linear_model.ElasticNet(alpha=alpha, l1_ratio=ratio)
        clf.fit(cross_X_train,cross_Y_train)
        y_pred = clf.predict(cross_X_valid)

        # Calculate RMSE and update minimum RMSE if possible
        RMSE = np.sqrt(mean_squared_error(cross_Y_valid, y_pred))
        if RMSE < minimum_RMSE:
            minimum_RMSE = RMSE
            minimum_alpha = alpha
            minimum_ratio = ratio
```

```
    counter +=1
    print counter
print "minimum RMSE is", minimum_RMSE
print "minimum alpha is",minimum_alpha
print "minimum ratio is",minimum_ratio


# #### D2. Elastic Net using PCA

# In[ ]:

# Set parameters to test
alphas = [.5,.1,.01,.001,.0001,.00001]
ratios = [.5,.1,.01,.001,.0001,.00001]

# Initialize minimums
minimum_alpha = 100
minimum_ratio = 100
minimum_RMSE = 100
for alpha in alphas:
    for ratio in ratios:

        # Fit model and predict on validation
        clf = linear_model.ElasticNet(alpha=alpha, l1_ratio=ratio)
        clf.fit(cross_X_train_transf,cross_Y_train)
        y_pred = clf.predict(cross_X_valid_transf)

        # Calculate RMSE and update minimum RMSE if possible
        RMSE = np.sqrt(mean_squared_error(cross_Y_valid, y_pred))
        if RMSE < minimum_RMSE:
            minimum_RMSE = RMSE
            minimum_alpha = alpha
            minimum_ratio = ratio

print "minimum RMSE is", minimum_RMSE
print "minimum alpha is",minimum_alpha
print "minimum ratio is",minimum_ratio




# coding: utf-8



# ## Load data
```

```
# In[ ]:

# df_train = pd.read_csv('sam_data/rdk_feat_eng_whole_df_train_orig_features.csv')
# df_test = pd.read_csv('sam_data/rdk_feat_eng_whole_df_test_orig_features.csv')
# df_train = pd.read_csv('final_data/FINAL_train.csv')
# df_test = pd.read_csv('final_data/FINAL_test.csv')
df_train = pd.read_csv('FINAL_interactions/FINAL_train_25_interactions.csv')
df_test = pd.read_csv('FINAL_interactions/FINAL_test_25_interactions.csv')
df_train.head()


# In[ ]:

df_test.head()


# In[ ]:

# Drop the 'smiles' and 'Id' columns
df_train = df_train.drop(['smiles'], axis=1)
df_test = df_test.drop(['Id'], axis=1)
df_test = df_test.drop(['smiles'], axis=1)

# Store gap values
Y_train = df_train.gap.values

# Delete 'gap' column
df_train = df_train.drop(['gap'], axis=1)
X_train = df_train.values
X_test = df_test.values
print "Train features:", X_train.shape, "Train gap:", Y_train.shape
print "Test features:", X_test.shape


# In[ ]:

# Split training data into training and validation sets as well as begin some k-fol
cross_X_train, cross_X_valid, cross_Y_train, cross_Y_valid = train_test_split(X_tra


# #### For classification purposes, round target values to nearest .5

# In[ ]:

# Round to nearest integer
```

```
# cross_Y_train_labels, cross_Y_valid_labels = np.round(cross_Y_train), np.round(cr
# Y_train_labels = np.round(Y_train)
# Round to nearest .5
cross_Y_train_labels, cross_Y_valid_labels = (((np.round(2*cross_Y_train)/2.0)-0.5)
Y_train_labels = (((np.round(2*Y_train)/2.0)-.5)/.5).astype(int)
# Round to nearest .25
# cross_Y_train_labels, cross_Y_valid_labels = (((np.round(4*cross_Y_train)/4.0)-.2
# Y_train_labels = (((np.round(4*Y_train)/4.0)-.25)/.25).astype(int)


# In[ ]:

print "'Training' features: ", cross_X_train.shape
print "'Validate' features: ", cross_X_valid.shape


# # GOAL:
#
# This notebook is set-up to chain together classification and regression methods.
#
# It's imperative that we get as accurate of a classifier as we can.
#
# Fingers crossed.

# ### First: Let's build a classifier that will adequately label the samples
#
# We'll start with Logistic Regression and try to fit the best model using a collec

# In[ ]:

get_ipython().run_cell_magic(u'time', u'', u'\nlogReg_training_acc = 0\nlogReg_test
clf_logReg=LogisticRegression(penalty="l2",C=c, solver=\'lbfgs\')\n
clf_logReg.fit(cross_X_train,cross_Y_train_labels)\n    training_acc = clf_logReg.s
test_acc = clf_logReg.score(cross_X_valid,cross_Y_valid_labels)\n
print c, test_acc\n    if logReg_test_acc < test_acc:\n        logReg_test_acc = te
logReg_training_acc = training_acc\n        best_logReg = clf_logReg')


# In[ ]:

logReg_training_acc = best_logReg.score(cross_X_train,Y_clf_train)
logReg_test_acc = best_logReg.score(cross_X_valid,Y_clf_valid)
print "Training Accuracy: %0.3f" % logReg_training_acc
print "Test Accuracy: %0.3f" % logReg_test_acc
```

```
# #### Concatenate predicted labels onto test/validation set

# In[ ]:

def adding_labels(feat_matrix,labels):
    '''Helper function that creates sparse binary array to concatenate to feat_matr
    # Create empty matrix
    added_cols = np.zeros((labels.shape[0],np.unique(labels).shape[0]))
    # Increment entry that corresponds to the sample having the specified label
    for ii in xrange(labels.shape[0]):
        added_cols[ii,labels[ii]] = 1
    # Concatenate label columns to feat_matrix
    feat_matrix = np.concatenate((feat_matrix,added_cols),axis=1)
    return feat_matrix


# In[ ]:

cross_X_train = adding_labels(cross_X_train,cross_Y_train_labels)

cross_X_valid = adding_labels(cross_X_valid,cross_Y_valid_labels)


# ## Now generating a Random Forest Regression

# In[ ]:

get_ipython().run_cell_magic(u'time', u'', u'ExtraTrees_RMSE = 100\nnum_estimators
\nfor n_estimators in num_estimators:\n\n    rf_reg = RandomForestRegressor(n_estim
rf_reg.fit(cross_X_train,cross_Y_train)\n    y_pred = rf_reg.predict(cross_X_valid)
RMSE = np.sqrt(mean_squared_error(cross_Y_valid, y_pred))\n    if RMSE < ExtraTrees
print n_estimators, RMSE\n        RF_RMSE = RMSE\n        RF_estimators = n_estimat
best_RF = rf_reg\n            \nprint "RandomForest with {0} estimators had RMSE of
')


# ## We're also going to tune a typical Linear Regression to have double coverage (

# In[ ]:

get_ipython().run_cell_magic(u'time', u'', u'lasso_RMSE = 100\nalphas = np.logspace
lasso_reg = Lasso(alpha=alpha)\n    lasso_reg.fit(cross_X_train, cross_Y_train)\n
y_pred = lasso_reg.predict(cross_X_valid)\n    \n    RMSE = np.sqrt(mean_squared_er
if RMSE < lasso_RMSE:\n        lasso_RMSE = RMSE\n        lasso_alpha = alpha\n
best_lasso = lasso_reg\n        \nprint "Lasso RMSE: {0} with alpha: {1}".format(la
```

```
# In[ ]:

# %%time
# ridge_RMSE = 100
# alphas = np.logspace(-4, 1, 30)

# for alpha in alphas:
#     ridge_clf = Ridge(alpha=alpha)
#     ridge_clf.fit(X_train_clf, cross_Y_train)
#     y_pred = ridge_clf.predict(X_valid_clf)

#     RMSE = np.sqrt(mean_squared_error(cross_Y_valid,y_pred))
#     if RMSE < ridge_RMSE:
#         ridge_RMSE = RMSE
#         ridge_alpha = alpha
#         best_ridge = ridge_clf

# print "Ridge RMSE: {0} with alpha: {1}".format(ridge_RMSE,ridge_alpha)


# In[ ]:

with open('final2_classifier_and_regressors.pkl','w') as f:
    pickle.dump((best_logReg, best_RF, best_lasso),f)


# In[ ]:

# with open('final_classifier_and_regressors.pkl','r') as fopen:
#     best_logReg, best_pcaExtraTrees, pcaExtraTrees_components, best_ridge = pickl

# After opening this, you may need to re-configure the test and training set, that


# ## Train on full training set, run on full test set

# Here we will train the classifier and the regressions

# In[ ]:

def write_to_file(filename, predictions):
    with open(filename, "w") as f:
        f.write("Id,Prediction\n")
        for i,p in enumerate(predictions):
            f.write(str(i+1) + "," + str(p) + "\n")
```

```
# ### Logistic Regression

# In[ ]:

get_ipython().run_cell_magic(u'time', u'', u'# Train classifier\nbest_logReg.fit(X_


# ### Random Forest

# In[ ]:

get_ipython().run_cell_magic(u'time', u'', u'# Train ExtraTrees Regressor\nbest_RF.


# ### Lasso/Ridge Regression

# In[ ]:

# # Train Ridge Regression
# best_ridge.fit(X_full_train_clf,Y_train)

# # Run Ridge Regressor
# ridge_pred = best_ridge.predict(X_test_clf)

# # Save the Ridge Predictions
# write_to_file("ridge_FINAL_TWK_10Feb.csv",ridge_pred)
# print "Completed Ridge Regression"

# Train Lasso Regression
best_lasso.fit(X_train_clf,Y_train)

# Run Lasso Regressor
lasso_pred = best_lasso.predict(X_test_clf)

# Save the Lasso Predictions
write_to_file("lasso_FINAL2_TWK_10Feb.csv",lasso_pred)
print "Completed Lasso Regression"




# In[ ]:
```

```
# Remove cross validation datasets
del cross_X_train, cross_X_valid, cross_Y_train, cross_Y_valid


# # V. Final Model Construction and Test Set Prediction
# 1. Read in training data
# 2. Read in test data
# 3. If using PCA, transform training and test data
# 4. Train model using training data
# 5. Predict output using test data
# 6. Write output to csv

# In[ ]:

df_test = pd.read_csv('FINAL_test.csv')


# In[ ]:

#delete 'Id' column
df_test = df_test.drop(['Id'], axis=1)
# Drop the 'smiles' column
df_test= df_test.drop(['smiles'], axis=1)
X_test = df_test.values


# In[ ]:

def write_to_file(filename, predictions):
    with open(filename, "w") as f:
        f.write("Id,Prediction\n")
        for i,p in enumerate(predictions):
            f.write(str(i+1) + "," + str(p) + "\n")


# In[ ]:

pca = PCA(n_components=60)
extraTrees_pca = ExtraTreesRegressor(n_estimators=100,n_jobs=2)
# extraTrees = ExtraTreesRegressor(n_estimators=25,n_jobs=2)

X_transf = pca.fit_transform(X_train)
X_test_transf = pca.transform(X_test)

tree_est_wPCA = extraTrees_pca.fit(X_transf, Y_train)
# tree_estimator = extraTrees.fit(X_train, Y_train)
```

```
pca_exTree_pred = tree_est_wPCA.predict(X_test_transf)
# exTree_pred = tree_estimator.predict(X_test)


# In[ ]:

#write to file
```