

Capstone Project Report - Machine Learning Engineer Nanodegree

Prarit Agarwal
Monday 5th August, 2019

Contents

1	Definition	1
1.1	Project Overview	1
1.2	Problem Statement	2
1.3	Metric	2
2	Analysis	3
2.1	Datasets Exploration	3
2.2	Exploratory Visualization	6
2.3	Algorithms and Techniques	8
2.4	Benchmark Model	10
3	Methodology	11
3.1	Feature Engineering	11
3.2	Data Preprocessing	12
3.3	Implementation	13
3.4	Refinement	15
4	Results	18
4.1	Model Evaluation and Validation	18
4.2	Justification	19
5	Conclusion	21
5.1	Free-Form visualization	21
5.2	Reflection	24
5.3	Improvement	25

1 Definition

1.1 Project Overview

This proposal is based on the [Kaggle competition](#) called “Predicting Molecular Properties” [1] hosted by the CHemistry and Mathematics in Phase Space (CHAMPS) group of researchers. The aim is to develop an algorithm to predict the strength of interatomic interactions called ‘scalar couplings’ in any given molecule. The knowledge of such scalar couplings is highly sought after and is extremely helpful in understanding the physical and chemical properties of these molecules [2]. In principle, it is possible to solve these scalar couplings through quantum mechanical computations. These computations involve solving the Schrodinger’s equation for a ‘many-body-system’, a problem that is known to be hard and time consuming owing to its computational intensiveness. At the same time, the scalar couplings are highly constrained by the requirement of invariance under translation and rotation of

the molecule. It therefore follows that they can only depend upon interatomic distances, the relative orientation of the atoms and the overall geometry of the molecule along with the different physical properties of the atoms themselves. Therefore, it should be possible to develop numerical models for these couplings which would greatly enhance our computational ability without a significant loss in precision when compared to an honest 'quantum mechanical computation'. This is also evident from the fact that chemist have already developed a reliable set of rules (for e.g. see [3] and [4]) which can be used to predict the coupling constants as long as the required precision is not too high. However, in applications such those in the pharmaceutical industry and material science, one often desires a higher precision than might be possible using these rules. Given the nature of the problem above, it is therefore natural to try to apply machine learning tools to this area and see if one can develop better models with more precise predictions.

1.2 Problem Statement

The CHAMPS group has provided explicit interatomic scalar couplings for 85003 molecules along with information about their 3d molecular structure. Using these to train models, one then has to make predictions for 45772 new molecules. Given that these scalar couplings can in principle take any real value, this is clearly a regression task. As is the case with any regression task, the goodness of the solution can be easily quantified in terms of the error. Additionally, if desired, one can also use the R_2 -score to gauge the predictability of the model.

1.3 Metric

As mentioned [here](#), submission are evaluated based on the log of the mean absolute error for each scalar coupling type type, which is then averaged over all types. More explicitly, it is given by

$$\text{score} = \frac{1}{T} \sum_{t=1}^T \log \left(\frac{1}{n_t} \sum_{i=1}^{n_t} |y_i - \hat{y}_i| \right) \quad (1.1)$$

where t runs over the different types of scalar couplings, i runs over the number of instances in the corresponding type and

- T = number of different scalar coupling types
- n_t = number of instances of type t
- y_i = prediction for the i -th instance
- \hat{y}_i = true value for the i -th instance

The competition webpage notes that, "For this metric, the MAE for any group has a floor of $1e-9$, so that the minimum (best) possible score for perfect predictions is approximately -20.7232".

A justification for using the Log of the MAE for different types of couplings as opposed to the total MAE is provided in the discussion [here](#). To understand the point being made here, let us note that the number of samples and the range of coupling constant values corresponding to each type varies quite a bit in the dataset. For e.g. there are 1510379 training instances of type '3JHC' while only 43363 training instances of type '1JHN'. This implies that if one is able to develop a model which predicts the

couplings of type '3JHC' extremely well while performing relatively poorly on the couplings '1JHN' (or vice-versa), the total MAE will still be skewed in the direction of '3JHC' due to the comparatively large number of samples of that type. Considering the Log of MAE of each type ameliorates this issue and balances things out such that a 1% decrease in MAE for one type provides the same improvement in score as a 1% decrease for another type. The metric chosen above therefore encourages us to focus equally on all types without getting biased by their relative dataset-size.

2 Analysis

2.1 Datasets Exploration

The dataset provided by the CHAMPS group is publicly available through this [link](#). As mentioned in the previous section, it consists of a training set with 85003 molecules and a test set consisting of 45772 other molecules. It has also been ensured that there is no overlap between the training and the test set. For all these molecules, CHAMPS has also provided the spatial coordinates for all the atoms in each molecule. This information is provided in a separate file called 'structures.csv'. This is to be treated as the only input to the models.

```
In []: # importing train, test and structures
train=pd.read_csv("train.csv", index_col='id')
test=pd.read_csv("test.csv", index_col='id')
struct=pd.read_csv("structures.csv")
```

The file "train.csv" contains all the training molecules. The first 11 entries in this file are:

```
In []: train.head(11)
```

Out []:

id	molecule_name	atom_index_0	atom_index_1	type	scalar_coupling_constant
0	dsgdb9nsd_000001	1	0	1JHC	84.8076
1	dsgdb9nsd_000001	1	2	2JHH	-11.2570
2	dsgdb9nsd_000001	1	3	2JHH	-11.2548
3	dsgdb9nsd_000001	1	4	2JHH	-11.2543
4	dsgdb9nsd_000001	2	0	1JHC	84.8074
5	dsgdb9nsd_000001	2	3	2JHH	-11.2541
6	dsgdb9nsd_000001	2	4	2JHH	-11.2548
7	dsgdb9nsd_000001	3	0	1JHC	84.8093
8	dsgdb9nsd_000001	3	4	2JHH	-11.2543
9	dsgdb9nsd_000001	4	0	1JHC	84.8095
10	dsgdb9nsd_000002	1	0	1JHN	32.6889

The first 10 rows of the training data set correspond to the same molecule, here labeled as "dsgdb9nsd_000001". We can find information about its constituent atoms by looking at the "structures" file and picking out the rows that have "dsgdb9nsd_000001" as the entry for the "molecule_name" column.

```
In []: # use pandas.DataFrame.loc to access the instances that satisfy a particular condition
# here we want to get the rows having "dsgdb9nsd_000001" as their "molecule_name"
struct.loc[struct['molecule_name']=="dsgdb9nsd_000001"]
```

```
Out []:
```

id	molecule_name	atom_index	atom	x	y	z
0	dsgdb9nsd_000001	0	C	-0.012698	1.085804	0.008001
1	dsgdb9nsd_000001	1	H	0.002150	-0.006031	0.001976
2	dsgdb9nsd_000001	2	H	1.011731	1.463751	0.000277
3	dsgdb9nsd_000001	3	H	-0.540815	1.447527	-0.876644
4	dsgdb9nsd_000001	4	H	-0.523814	1.437933	0.906397

We see that "dsgdb9nsd_000001" consists of 5 atoms: 1 Carbon (C) atom and 4 Hydrogen (H) atom. It is reasonable to guess that this is a CH_4 molecule. Note that in the CH_4 molecule, all the 4 hydrogen atoms are equivalent, so we would expect their properties to be the same. This is further confirmed from the training data by noticing that the coupling constant between any two hydrogen atoms is almost the same, being equal to ~ -11.25 . Similarly, the coupling constant between the carbon atom and any of the hydrogen atoms is approximately ~ 84.80 .

Having established that "dsgdb9nsd_000001" is CH_4 , we can use this to understand the entries in the column labeled "type" in the "train" file: we notice that the format of the entries under this column is "nJAtom1Atom2" for e.g., 1JHC, 2JHH, 1JNH etc. By comparing to the molecular structure of CH_4 as shown [here](#), we realize that for the coupling of type "nJAtom1Atom2", that there are "n-links" between Atom1 and Atom2, or in other words, one has to take n-steps as one goes from Atom1 to Atom2. For e.g. there is a single link between the carbon atom and any of the hydrogen atoms. It therefore implies that we need exactly 1 step to go from the carbon atom to any of the hydrogen atoms or vice-versa. Thus the "type" should be 1JHC which is indeed the case for "dsgdb9nsd_000001". Similarly, in CH_4 , we will have to take exactly 2 steps to go from one hydrogen atom to another hydrogen atom. Thus the corresponding type should be 2JHH for any pair of hydrogen atoms. Again, this is the case for "dsgdb9nsd_000001".

We also notice that the symmetry between the hydrogen atoms of CH_4 requires all their coupling constants to be exactly the same. However, we notice that they are equal only upto 2 decimal places. Thus, we shouldn't expect the machine learning algorithms trained on this data to produce results that are accurate at more than 2 decimal places.

Let us also check for missing values in the data.

```
In []: # Checking for missing values in train.csv
train.isnull().values.any()
```

```
Out []: False
```

```
In []: # Checking missing values in test.csv
test.isnull().values.any()
```

```
Out []: False
```

```
In []: # Checking missing values in structures.csv
struct.isnull().values.any()
```

```
Out[]: False
```

From above, we see that there are no NaN's in either of the files.

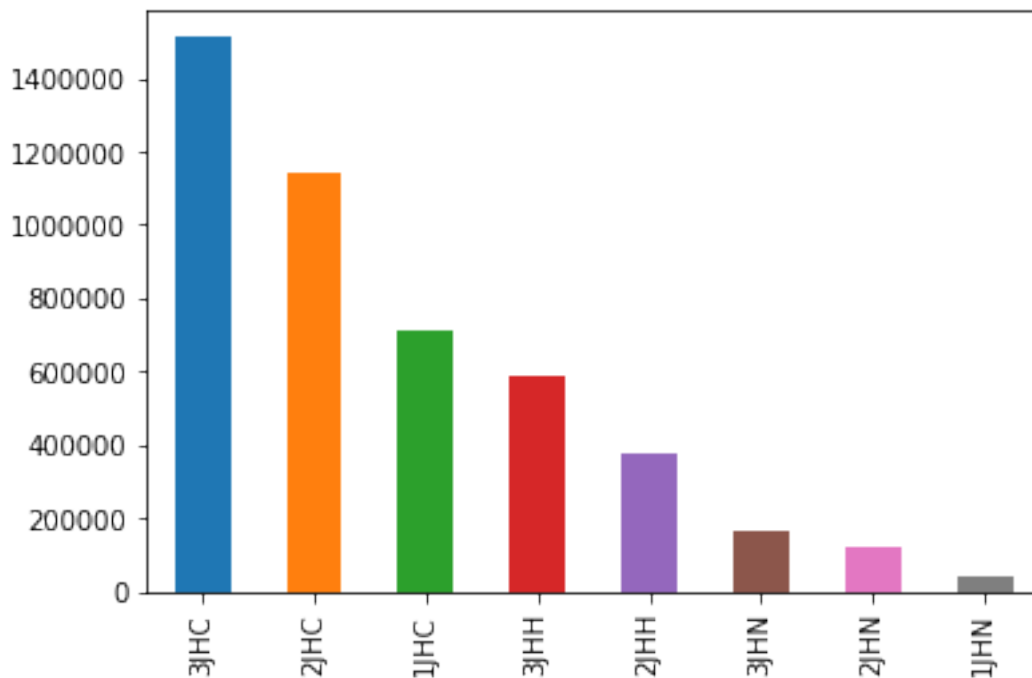
The only categorical feature is 'type'. Let us check how many different values can it take.

```
In []: train['type'].value_counts()
```

```
Out[]: 3JHC      1510379
2JHC      1140674
1JHC       709416
3JHH       590611
2JHH       378036
3JHN       166415
2JHN       119253
1JHN       43363
Name: type, dtype: int64
```

This can also be visualized through the following bar-chart:

```
In []: plt.figure()
train['type'].value_counts().plot(kind='bar')
plt.show()
```



From above we see that there are 8 different 'types' of couplings: 1JHC, 1JHN, 2JHC, 2JHN, 2JHH, 3JHC, 3JHN, 3JHH. The coupling type 3JHC is the most frequent and 1JHN is the least frequent of these.

We also found that the various different kinds of atoms present in any of the molecules appearing in the train/test set are always a subset of $\{C, H, N, O, F\}$ i.e Carbon, Hydrogen, Nitrogen, Oxygen and Fluorine respectively:

```
In []: # Extracting the different atoms that appear in the database
atoms=struct['atom'].unique()

In []: print(len(atoms))

5

In []: print(atoms)

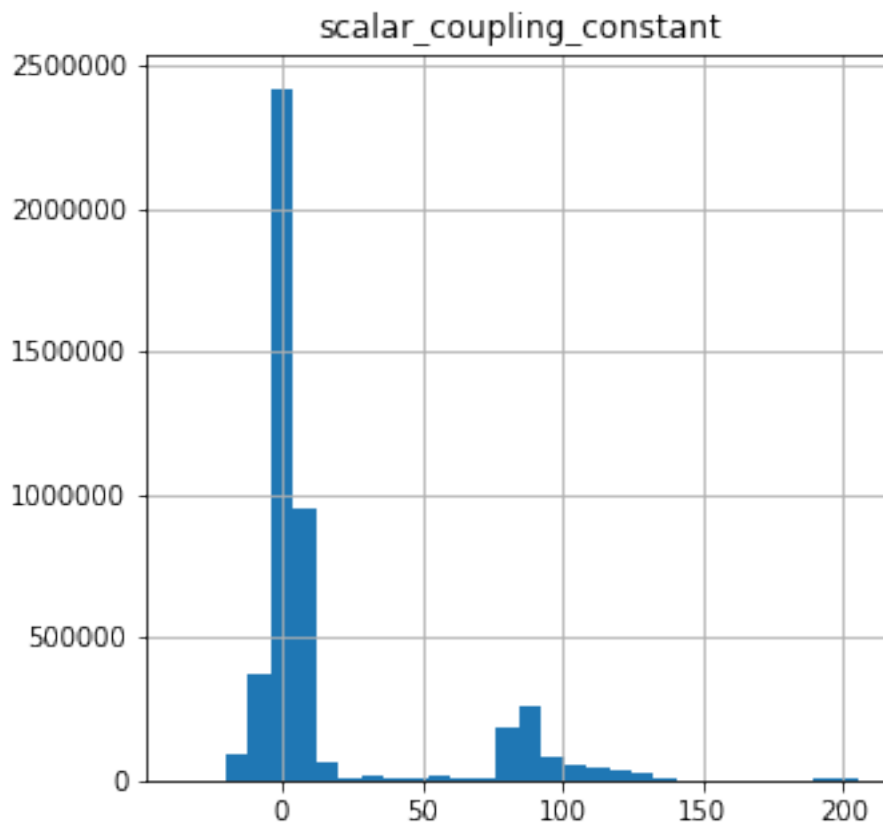
['C' 'H' 'N' 'O' 'F']
```

CHAMPS has also provided additional data namely, *dipole moments*, *magnetic shielding tensors*, *mulliken charges*, *potential energy* and *scalar coupling contributions* for the molecules in the training set only. This data is not available for the molecules in the test set. Thus, if one intends to use this information, then they will have to separately model them first and then use them as meta-features for the molecules in the test set. However, as is well known to anyone with a little physics/chemistry background and is also mentioned in literature (see for e.g. [5]), a first principles quantum mechanical calculation of these coupling constants only requires information about the relative positions of the various atoms in the molecules. Therefore, we will try to come up with a model which does not use these meta-features and ignore them for now.

2.2 Exploratory Visualization

Let us now consider distribution of target variable i.e. the coupling constant values. We can do this by plotting a histogram as follows:

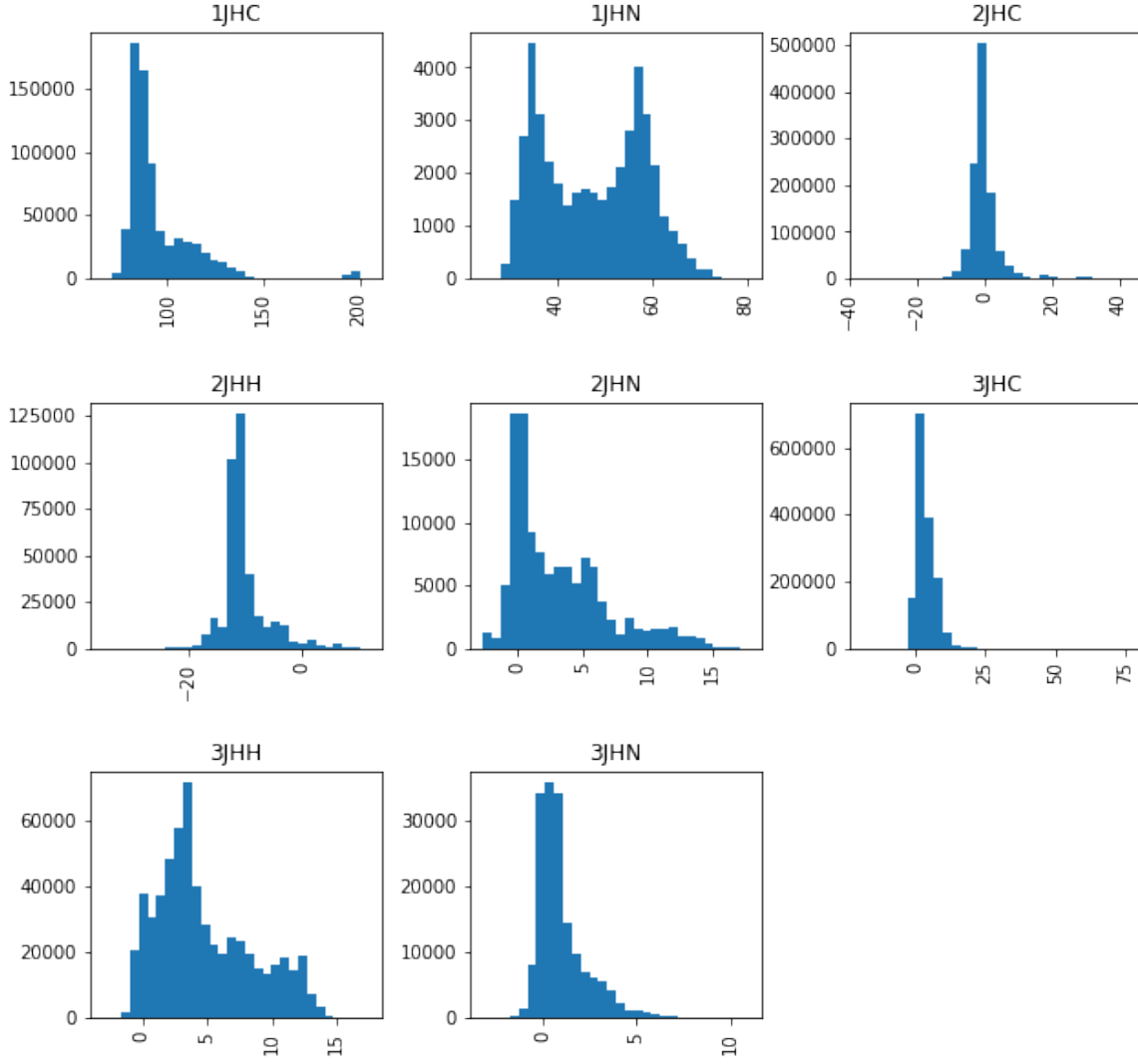
```
In []: hist_all=train.hist( column='scalar_coupling_constant', figsize=(5,5), bins=30)
```



We notice that while most of the values lie in the range ~ -25 to 25 , there are still quite a significant number that additionally lies in the range ~ 75 to 150 along with a small number of values that are distributed around 200 .

A more refined understanding of the distribution of the target variable can be obtained if we group them by their coupling-type. The corresponding distributions are given by the following histograms

```
In []: # see this link for tips on using pandas histogram:
# https://mode.com/example-gallery/python_histogram/
hist_grouped_by_type=train.hist( column='scalar_coupling_constant',
                                by='type', figsize=(10,10), bins=30)
```

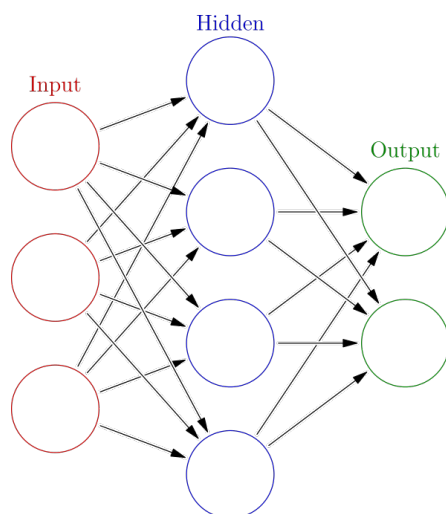
We notice that the distribution of the coupling constants varies quite a bit across the different types. For e.g. the distribution is more or less gaussian for type ‘2JHC’ and ‘2JHH’, however for the type ‘1JHN’ it seems to contain two overlapping gaussians. Similarly, the range of values taken by the coupling constant also varies quite a bit with the coupling type. For e.g., the coupling constants in type ‘2JHC’ seem to be normally distributed between ~ -15 to 15 with a small number of outliers around ~ 20 and also around ~ 30 . On the other hand, the values for type ‘1JHC’ appear to mostly lie between ~ 50 to ~ 150 with a few outliers around 200 . Similarly, the values for type ‘2JHH’ are mostly negative while that for type ‘3JHH’ are mostly positive. This suggest that it is perhaps best to build independent models for each type, such that they are able to best capture the individual characteristics that are at play in each individual type.

2.3 Algorithms and Techniques

Given that the explicit values for the scalar couplings of different types have different distributions, we propose to train an individual model for each type. We also wish to use neural networks to

create our models. As mentioned in section 1.1, translational and rotational invariance requires that the molecular properties should only depend upon the interatomic distances and angles. These can easily be extracted given the spatial coordinates of all the atoms in each molecules. Having done this, we will feed this information into our model and train them appropriately, using regularization techniques such as batch-normalization, dropout and early stopping.

Our choice for using neural networks is guided by the fact that they are highly versatile and are able to model a huge variety of non-trivial relationships between the input data and the target variable. In layman's term, a neural network consists of rows of 'neuron' stacked on top of each other. The input to any neuron is an ordered-set of values. The neuron takes a weighted sum of these and passes them through a prescribed non-linear function. The value thereby produced by this non-linear function forms the output of the neuron. Stacking of rows of neurons on top of each other, further allows to scramble the input data in a myriad of complex ways thereby allowing the neural network to be able to capture almost any functional relationship between the input data and the target. Thus the complexity of the neural network is controlled by both the number of neurons in hidden layers as well as the number of hidden layers. Pictorially, a neural network can be represented as in the following figure (with each node in the graph corresponding to a neuron) :



The above picture, taken from [wikipedia](#) depicts a neural network with a single hidden layer, however in general, there can be multiple such hidden layers. In the training phase, an optimal set of weights and biases are found that will minimize the error in the values predicted by the neural network when compared those provided in the training set.

At the same time, we also wish to apply ensembling techniques to improve our predictions. To this end, for each type of scalar coupling, we will train a couple of different neural networks (with different configurations/hyper-parameters) and take the weighted average of their predictions as our final prediction. The weights to be applied to each individual model will be based on their R_2 -score for the validation set i.e. the predictions from the model with a higher R_2 -score will be given a proportionately higher weight.

The rational behind the above described ensembling is that models with different architecture will most likely capture somewhat different aspects of the input data and therefore represent slightly different functional relationships between the input and the target. Aggregating over the results of these models is therefore expected to give us a more comprehensive and robust relationship between input data and the target while at the same time reducing any over-fitting done by the individual models. At the same time, since the cross-validation R_2 -score of any individual model gives us a quantitative measure of its performance on previously unseen data, therefore, it is natural to weight the prediction of each model by its R_2 -score on the validation set.

In the end, let us however note a drawback of neural networks: the high degree of complexity of the neural network also means that it is usually not possible to describe the model generated by a neural network in the form of a simple mathematical equation/rules.

2.4 Benchmark Model

A benchmark model has already been suggested by the competition sponsors. This can be found [here](#). It is based on engineering a single feature i.e. the distance vector between the atoms involved in the coupling. Then scikit-learn's RandomForestRegressor, along with a GroupKFold splitting (`n_splits=3`) is used to predict the scalar coupling constant. It Note that the dataset is split into folds by molecules. The RandomForestRegressor is configured with the following parameter:

```
n_estimators = 250, max_depth = 9, min_samples_leaf = 3, n_jobs = -1
```

We implemented this benchmark in the jupyter notebook (included in the git-hub repository) called 'CHAMPS_Benchmark.ipynb'. The benchmark values for mean absolute errors on the training set data (for each coupling type) are:

	type	mean_absolute_error
0	1JHC	4.901738
1	2JHH	1.357144
2	1JHN	2.721465
3	2JHN	2.555075
4	2JHC	2.427572
5	3JHH	1.886179
6	3JHC	2.099637
7	3JHN	0.914532

Upon submitting the test set values produced by the benchmark model to Kaggle, the score obtained is 0.750 (recall according to the competition metric the lower the score, the better it is).

Your most recent submission				
Name	Submitted	Wait time	Execution time	Score
atomic_distance_benchmark.csv	a few seconds ago	0 seconds	35 seconds	0.750
Complete				
Jump to your position on the leaderboard ▾				

3 Methodology

3.1 Feature Engineering

The data provided by the CHAMPS group gives us the position coordinates of each atom in every molecule. However, translational and rotational invariance of the scalar couplings imply that they should only be dependent on the inter-atomic distances and angles within each molecule. These can be easily engineered from the given data.

We separately extracted the interatomic distances between all possible pairs of atoms in each molecule and stored in a file called 'CHAMPS_rel_pos.csv'. The python implementation for this step is given in the jupyter notebook called `interatomic_distances.ipynb`.

A little thought reveals that the coupling constant should decrease as the interatomic distances decrease. This is most easily seen from the fact that if two atoms are placed at a large distance from each other, then they are effectively isolated from each other's influence and hence the coupling between them should have a zero magnitude. We will therefore engineer features corresponding to inverse powers of the interatomic distances. These will be: distance^{-1} , distance^{-2} , distance^{-3} , distance^{-4} , distance^{-5} , distance^{-6} . While we can certainly consider features distance^{-n} for $n > 6$, our decision to limit ourselves to $n \leq 6$ is two-fold:

- As n becomes large, distance^{-n} rapidly becomes zero for all atomic pairs, thereby reducing the variance of such a feature. Thus increasing n to very high values has diminishing returns.
- The van der Waals electrostatic interaction between atoms has a distance^{-6} dependence. Given that van der Waals interaction is quite important in understanding various molecular properties, we definitely wish to keep this as one of our feature.

Additionally, it is well known that the neighboring atoms have quite a strong effect on these coupling constants (for e.g. see [2]). We will therefore also include distances to the three nearest atoms of each kind i.e. C, H, N, O & F. To be more precise, we will include the three nearest atoms of each kind for each of `atom_0` and `atom_1`. This will therefore give us $3 \times 5 \times 2 = 30$ new features. A similar idea was also suggested in [this](#) kaggle kernel, whose implemetation was greatly improved in [this](#) kernel. Since, once again the effect of nearest neighbors should decrease with their distance from the atoms

in the coupling, thus we will once again consider their inverse powers. As before, we will limit the inverse power to be less than or equal to 6.

An important point to note, regarding the above mentioned nearest neighbors is that many of the molecules might not contain an atom of particular kind, or it might contain less than three atoms of a particular kind. For e.g. CH_4 does not contain the atoms N, O, F . Also, it only contains a single C atom. A simple way to deal with these ‘missing atoms’ is to think of them as being present at an infinite distance from the molecule (this idea was also used in [5]). For practical purposes, instead of using infinity, we will simply use the value 1000 for their corresponding distance. This is certainly much much larger than any of the other interatomic distances and should suffice for us.

Upon going through the steps mentioned in this section, we will end up with a set of 217 features i.e. distance between `atom_0` and `atom_1` along with its 6 inverse powers, distance of three nearest neighbors of each atomic kind with respect to `atom_0` and their 6 inverse powers and distance of three nearest neighbors of each atomic kind with respect to `atom_0` and their 6 inverse powers, giving us $7 + 3 \times 5 \times 7 + 3 \times 5 \times 7 = 217$.

3.2 Data Preprocessing

Note that the range of different features engineered above is quite different from each other. This is simply because distance^{-2} will obviously have a different range than distance^{-3} and so on. This difference in range of different features can adversely effect the learning process of a neural network, hence we will use scikit-learns StandardScaler to rescale all the features. This will set the mean of all features to zero and set their variance to 1. Also, since we will be building independent models for each coupling type, we will implement this feature rescaling separately for each coupling type.

Further, we used pca reduction to reduce the dimension of our feature space. To gauge the number of components to keep, we computed their explained variance ratio and kept the minimum number of components required to explain 99.99% of the total variance in the original feature space. Again we did this separately for each individual type. For all the different type, the number of pca component that got chosen in this way were between 74 and 78. As an aside, we also wish to note, as is also discussed [here](#), a standard way to choose the appropriate no. of pca components is by plotting their cumulative explained variance ratio and choosing the point where the plot plateaus off. However, we wanted the computer to choose the number of components automatically given that doing this manually for each type would be a little tedious. At the same time, it is not simple for the computer to detect where the plot plateaus, so as a trade-off, we just implemented a function that chose the no. of components needed to explain 99.99% of the variance.

From the EDA done above, one should also note that the target values i.e. scalar coupling constants take values over large range of numbers from ~ -25 to ~ 200 . This large range of values will make training a very slow process. For, example, see [this](#) blog-post. This was certainly the case for us. In our attempt to improve our models, we found that training was much much faster once we used the

StandardScaler to scale the target values. The predictions for the test set were simply obtained by applying an inverse transformation to the results produced by our models.

3.3 Implementation

To begin with, we wish to point out that even though the raw data provided by the CHAMPS group is not too large, the engineered features were very memory intensive such that it was impossible to run the codes on a laptop. We therefore ran our codes on Google Colab. Therefore, for the purpose of verification of , we will be attaching only a small fraction of the sample data. The codes can then be run on any generic computer, though the immediate results produced during such a verification might not be representative of our true results.

Our implementation can be divided into three modules, each of which was carried out in its own dedicated jupyter notebook¹. These are:

1. `interatomic_distances.ipynb`: The `structures.csv` file provided by the CHAMPS group, contains the (x, y, z) position coordinates of each atom in all the molecules. The jupyter notebook `interatomic_distances.ipynb`, consists of a single function that does the following:

- Read `structures.csv` into a DataFrame called 'structures'
- Use the above DataFrame to obtain all possible pairs of atoms in every molecules².
- Compute the difference between the coordinates of atoms in each pair. This gives the relative position vectors for each pair
- Compute the norm of the above position vectors to obtain the inter-atomic distance for each pair of atoms.
- Save the above results in a file called `CHAMPS_rel_pos.csv`

2. `nearest_neighbors.ipynb`: Let us call the atoms appearing in any atomic pair as `atom_0` and `atom_1` respectively. These pairs can be classified by the type of their coupling constant. In `nearest_neighbors.ipynb`, we compute the distances to three nearest neighbor atoms (of each type i.e. C, H, N, O and F) of `atom_0`. We do this separately for each type of coupling constant and store the results independent csv files. These follow the nomenclature '`CHAMPS_angles_type_atom_0.csv`', where *type* corresponds to the type of coupling constant. We repeat the above exercise for `atom_1` and store the results in '`CHAMPS_angles_type_atom_1.csv`'.

3. `predicting_scalar_couplings.ipynb`: This is the main module in our implementation. Here, we create a validation set by separating 20% of randomly selected molecules from all the molecules in the training set. These are then used to train the DNNs used for this project. Some of important functions defined in this notebook are:

¹We provide extensive annotations in the jupyter notebooks for all the steps involved.

²This can be easily done by using `pandas.DataFrame.merge` to merge two copies of the 'structures' DataFrame, using the 'molecule_name' as the key to join on.

`neighbor_info`: This takes the type of coupling as its sole input. Corresponding to the type of coupling passed to it, the function then reads the files `'CHAMPS_angles_type_atom_0.csv'` and `'CHAMPS_angles_type_atom_1.csv'` that contain information about the distance of `atom_0` and `atom_1` from their respective neighboring atoms. Following this, it computes their inverse powers as was mentioned in section 3.1. It returns a dataframe containing all these features, as its output.

`feature_processing`: This takes the DataFrame produced by `neighbor_info` and applies StandardScaler followed by PCA to these features. Note that it automatically computes the number of components that should be kept in order to explain 99.99% of the total variance in the training set data. For the molecules in the validation set, it simply transforms the corresponding features using the StandardScaler and PCA that were previously fitted to the training set features. The processed features are stored in arrays called `X_train` and `X_val` respectively. The corresponding values for the target variable are stored in `y_train` and `y_val` respectively. It returns, `X_train`, `y_train`, `X_val`, `y_val`, the fitted StandardScaler and PCA, along with the number of pca components, as its output.

`test_features`: This takes the type of coupling constants along with the instances of StandardScaler and PCA fitted to the corresponding training set feature space as its input. It uses these to transform the test set features and returns the processed test set features as its output.

`transform_target`: This function also takes the type of coupling as its sole input. It then `fit_transform` the training set values for the target variable using a new instance of StandardScaler. It returns this StandardScaler as its output. Note that in principle we could have created a pipeline that first applies scikit-learn's PowerTransformer followed by the StandardScaler to the target values. However, we found that for some reason, PowerTransformer produced a singular transformation for coupling type '1JHC'. This resulted in a number of nans. Therefore, we only applied the StandardScaler and not the PowerTransformer.

`model_builder`: This function creates and returns a DNN according to given specifications. Note that, the output layer will always have a single neuron without any activation. Also, we will use an l_1 regularization for kernel weights. The function inputs are as follows:

`n_features`: no. of input features

`hidden`: list of number of neurons in each hidden layer i.e. `[n_hidden1, n_hidden2, ...]`

`kernel_reg`: regularization parameter for kernel weights; default value will be 0.01

`regularizer`: 'Dropout' vs 'BatchNormalization'; Default will be BatchNormalization

`dropout_rate`: only used when regularizer is 'Dropout'; Default value is 0.2

`activation`: default is 'relu'

`R2_score`: Note that scikit-learn's `r2_score` library can not be passed to tensorflow for the puposes of computing R_2 scores of the results produced by the neural network. Therefore, we had to write a separate function for this purpose. It takes the true values and predicted

values for the target variable as its input and return the corresponding R_2 score for the predictions.

Along with the functions listed above, there are a few other helper functions which help create an ensemble of DNNs for each type and process the results produced.

3.4 Refinement

To begin with, we found that for some reason, upon using Dropout-layers the resulting DNNs did not perform very well and mean absolute errors for their predictions were pretty high. Due to this, we only used BatchNormalization layers in our DNNs.

Also, in our initial attempt we did not apply any PCA reduction to our features. All the features were mere scaled by an instance of a StandardScaler, so as to make sure all of them have unit variance. We then created a separate DNN for each coupling-type trained them appropriately.

The first DNN that we tried consisted of 10 hidden layers having 512,256,256,128,128,64,64,32,8,2 neurons respectively. For the coupling type '1JHC', this produced a mean absolute error of 0.904 on the validation set, with the corresponding R_2 score being 0.982. This is already quite good.

We tried to improve this by increasing the number of neurons in some of the layers. Thus our second DNN also consisted of 10 layers with 768,512,512,256,128,128,64,32,8,2 neurons respectively. For the coupling type '1JHC', this produced a mean absolute error of 0.879 on the validation set, with the corresponding R_2 score being 0.983. This is a small improvement over the previous model.

Next we tried to include more layers in the DNN. Thus we built a DNN with 13 hidden layers having 768, 768, 512, 512, 256, 256, 128, 128, 64,64, 32, 8, 2 neurons. For the coupling type '1JHC', this produced a mean absolute error of 0.851 on the validation set, with the corresponding R_2 score being 0.983. This is again a marginal improvement on the previous two models.

Finally, we tried a model with a much much larger number of neurons. This consisted of 13 layers with 1024, 1024, 768, 768, 512, 512, 256, 128,64,64,32,8,2 neurons respectively and produced a mean absolute error of 0.846 and R_2 score of 0.984 on the validation set for the coupling type '1JHC'. Again this did not give us a significant improvement over the previous scores.

DNN	no. of hidden layers	neurons in hidden layers	(mae, r_2) on 1JHC
1	10	512,256,256,128,128,64,64,32,8,2	(0.904, 0.982)
2	10	768,512,512,256,128,128,64,32,8,2	(0.879, 0.983)
3	13	768, 768, 512, 512, 256, 256, 128, 128, 64,64, 32, 8, 2	(0.851, 0.983)
4	13	1024, 1024, 768, 768, 512, 512, 256, 128,64,64,32,8,2	(0.846, 0.984)

In hindsight, the fact that even after considering more and more complex DNNs, we were unable to obtain a significant improvement in our scores is related to the fact the the R_2 score of the fit produced by our first model is already pretty high. It can already explain 98.2% of the variance in the validation set target variable. This implies, that the first model is able to almost obtain the best possible fit with the given features. Thus building model with more layers or larger number of neurons do not change the score much.

We then tried to see if taking the weighted average of the predictions produced by the individual models would improve the results. The R_2 score of each presents a natural choice to use as weights of the corresponding predictions. This is because the model with a higher R_2 score produces a better fit to the target variable and hence should be given preference over others. Upon doing so, we obtained a mean absolute error of 0.745 and an R_2 score of 0.996 for the validation set of coupling type 1JHC. This is $\sim 11\%$ improvement over our best model, clearly, demonstrating, the usefulness of ensembling techniques. We therefore implemented this for all the other coupling types also.

We therefore built 3 to 4 different DNNs for each coupling type and took a weighted average of their predictions. The respective final scores for each coupling type are listed in the table below.

coupling type	(mae, r_2) for the ensemble of DNNs
3JHN	(0.175, 0.921)
3JHC	(0.585, 0.876)
3JHH	(0.223, 0.985)
2JHC	(0.466, 0.962)
2JHN	(0.201, 0.985)
2JHH	(0.164, 0.993)
1JHN	(0.345, 0.998)
1JHC	(0.745, 0.996)

Upon submitting this to kaggle the Log mae score for the CHAMPS test set was -1.164.

Next, we tried to see how pca reduction will affect our predictions. The number of pca components required to explain 99.99% of the feature-variance for each type is listed below.

coupling type	no. of pca components
3JHN	78
3JHC	78
3JHH	76
2JHC	77
2JHN	75
2JHH	76
1JHN	75
1JHC	75

We found that the application of pca reduction alone did not significantly affect our results for coupling types 1JHC and 2JHN whose mean absolute error (after ensembling) for the validation set came out to be 0.736 and 0.163. However, the results improved drastically for coupling type 3JHC whose mean absolute error for validation set improved to 0.313. These results are tabulated below.

coupling type	(mae, r_2) before pca-reduction	(mae, r_2) after pca-reduction
1JHC	(0.745, 0.996)	(0.736, 0.996)
2JHN	(0.201, 0.985)	(0.163, 0.993)
3JHC	(0.585, 0.876)	(0.313, 0.968)

The fact that pca-reduction did not improve the results for coupling types 1JHC and 2JHN can again be understood from the fact that even before pca-reduction their corresponding R_2 -score was very high, showing that an good fit with respect to the available feature space had already been achieved, leaving very little room for improvement. On the other hand, for 3JHC the r_2 score before pca-reduction was merely 0.876, indicating that there was plenty of room for improvement. PCA reduction therefore allowed the DNNs to hone in on less over-fitted models. Given that the r_2 scores for all other coupling types (other than 3JHC) are already pretty high, we did not expect any further improvement for them by merely applying pca reduction.

Following the discussion in [this](#) blog, we noticed that the large range of values taken by our target variable can adversely affect training. We therefore tried to improve this by scaling the target values through another StandardScaler. We also tried to make the distribution of target variables more Gaussian by applying scikit-learn's [PowerTransformer](#), but it resulted in a singular inverse transformation for the case of coupling type 1JHC, producing nans in our results. We therefore decided not to use the PowerTransformer.

To our surprise, scaling the target values as mentioned above, greatly improved our training times, though the final scores did not change significantly. As a quantitative demonstration of this improvement, let us consider the DNN with 13 hidden layers consisting of 768, 768, 512, 512, 256, 256, 128, 128, 64, 64, 32, 8, 2 neurons. For coupling type 1JHC, before the scaling of target values, this DNN

took about 838 epochs to converge to its lowest possible error. Each epoch took about 7 seconds to be executed. On the other hand, once the target values were appropriately scaled, the DNN was able to converge to its lowest error within a mere 194 epochs, with each epoch taking about 6s.

4 Results

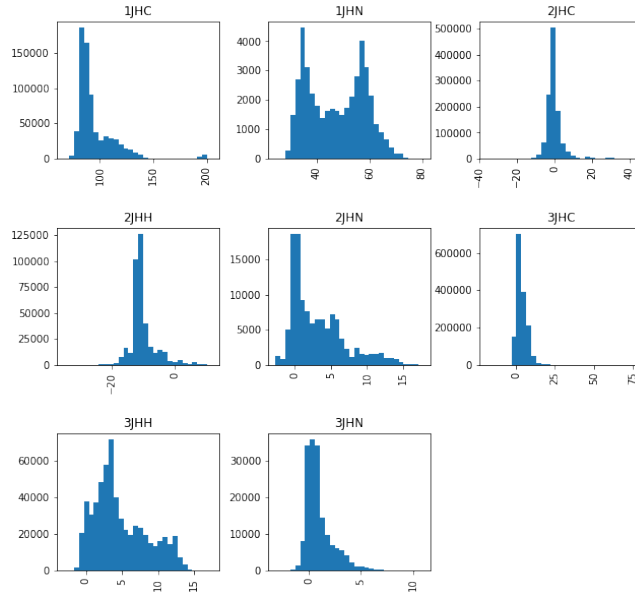
4.1 Model Evaluation and Validation

The training set contained data about 85003 molecules. We randomly chose 20% of these and set them aside for cross validation purposes. This ensured that the validation data for all the coupling types came from the same set of molecules. The DNN Checkpointer was set to save the DNN weights only if the mean absolute error for cross-validation set improved, irrespective of the improvement in the mean absolute error for the training set. This ensured that our DNNs did not produce a hugely overfitted model. We also used early stopping to control for overfitting and stopped training if the validation loss did not improve by at least 0.01 after 100 epochs. As was discussed in section 3.4, the different DNN configurations that we tried did not differ too much in their scores, but taking a weighted average of their results improved the final predictions to some extent. We therefore chose to fit about 3 to 4 different DNNs for each coupling type and took the weighted average of their results to produce our final predictions. On top of it, apply pca to feature space, further reduced overfitting. Similarly, scaling the target variable helped reduce training time. The final scores (with respect to the ensembled predictions) for the training and validation set are tabulated below.

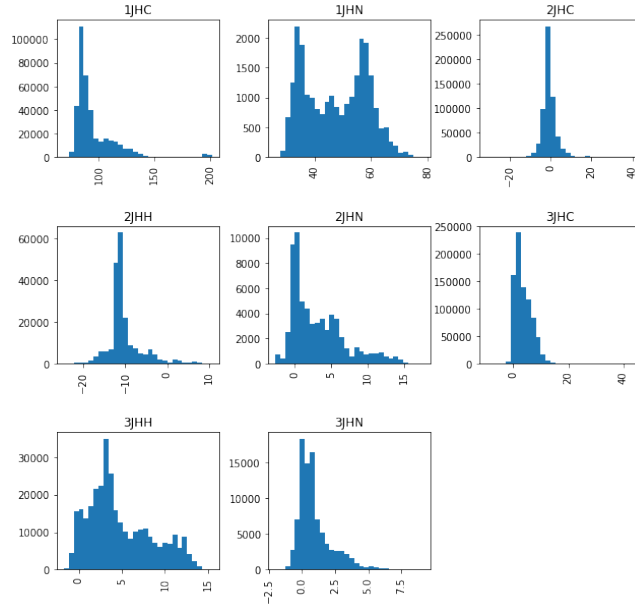
type	(mae, r_2) for training set	(mae, r_2) for validation set
3JHC	(0.110, 0.995)	(0.287, 0.972)
3JHN	(0.044, 0.998)	(0.129, 0.962)
3JHH	(0.066, 0.999)	(0.149, 0.995)
2JHC	(0.095, 0.999)	(0.245, 0.990)
2JHN	(0.084, 0.999)	(0.163, 0.993)
2JHH	(0.078, 0.999)	(0.141, 0.995)
1JHN	(0.307, 0.999)	(0.400, 0.997)
1JHC	(0.367, 0.999)	(0.643, 0.997)

We can see that though the scores for training set are slightly better than those for the validation set, the difference in the scores is not too large, thereby giving us the confidence that there is not too much overfitting and therefore our models will do a good job even on previously unseen data.

As a final check on the performance of our models, we graphed the distribution of predictions made on the test data provided by the CHAMPS group. If our models are to be trusted, then the predictions on the CHAMPS test data should have a similar distribution as the target values provided for the training data. The distribution of the target values for the training set is



While, the distribution of predictions for the test set is



We see that the two sets of distributions are quite similar between the training set and the test set. Though this is not a very strong test but it is still an important test for the trustability of our models and adds to our confidence that indeed we are on the right track towards finding better and better solutions to the problem.

4.2 Justification

Let us now compare the performance of our models with respect to that of the benchmark model provided by the CHAMPS group. The mean absolute errors for each coupling type for the benchmark model were reported in section 2.4. We will reproduce them in the following table and compare them with the validation set mean absolute error for our models.



type	mae for benchmark	mae for our models
3JHC	2.100	0.287
3JHN	0.915	0.129
3JHH	1.886	0.149
2JHC	2.428	0.245
2JHN	2.555	0.163
2JHH	1.357	0.141
1JHN	2.722	0.400
1JHC	4.902	0.643

We see that our models vastly out-perform the benchmark model for each coupling type. This is also evident from the log mae scores obtained on the CHAMPS test data. For the benchmark model the log mae was 0.750,

Your most recent submission				
Name	Submitted	Wait time	Execution time	Score
atomic_distance_benchmark.csv	a few seconds ago	0 seconds	35 seconds	0.750
Complete				
Jump to your position on the leaderboard ▼				

while the predictions produced by our models obtained a log mae score of -1.469 and a leaderboard position of 424 (as of 2nd Aug'19).

Your most recent submission				
Name	Submitted	Wait time	Execution time	Score
submission_v2.csv	a few seconds ago	0 seconds	36 seconds	-1.469
Complete				
Jump to your position on the leaderboard ▼				

424	Prarit		-1.469	3	2m
Your Best Entry ↑ You advanced 260 places on the leaderboard! Your submission scored -1.469, which is an improvement of your previous score of -1.164. Great job!					
					Tweet this!

While we believe that there is still quite a bit of room for improvement, but given the low values for mean absolute errors, it can be safely concluded that our models have largely solved the problem, if not completely.

5 Conclusion

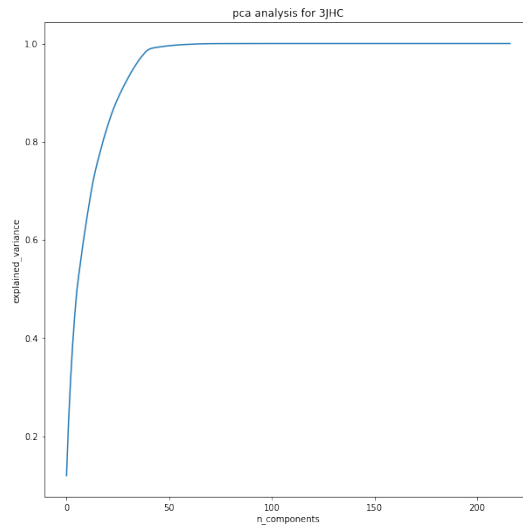
5.1 Free-Form visualization

One of the best ways to estimate the number of pca components to keep is to plot a graph of cumulative explained variance ratio versus number of pca components. The code to do this is as given below:

```
In []:
pca=PCA()
pca.fit(X_train)
cumulative_explained_variance=np.cumsum(pca.explained_variance_ratio_)
plt.figure(figsize=(10,10))
plt.plot(cumulative_explained_variance)
plt.xlabel('n_components')
plt.ylabel('explained_variance')
plt.title('pca analysis for {}'.format(tp))
plt.show()
```

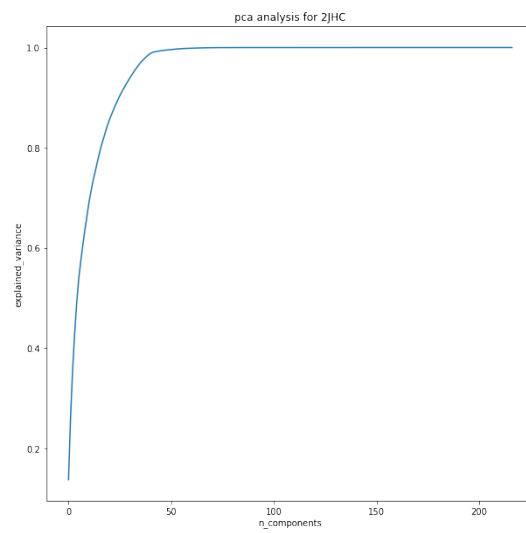
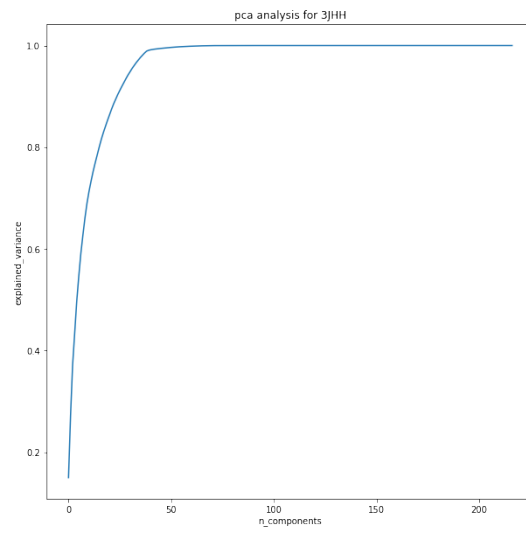
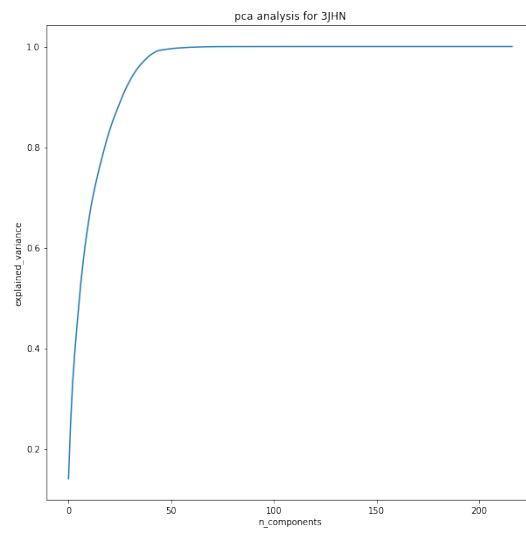
The number of pca components one should keep then corresponds to the point where this graph plateaus-off. Let us demonstrate how this applied to our case.

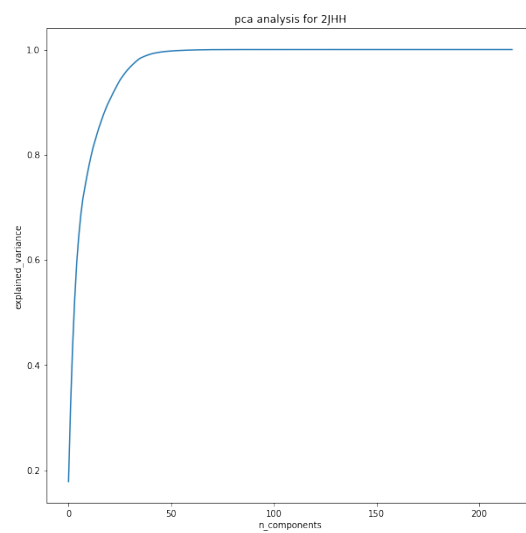
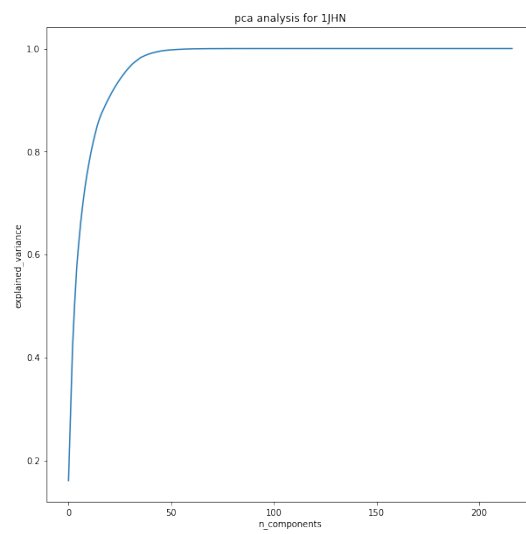
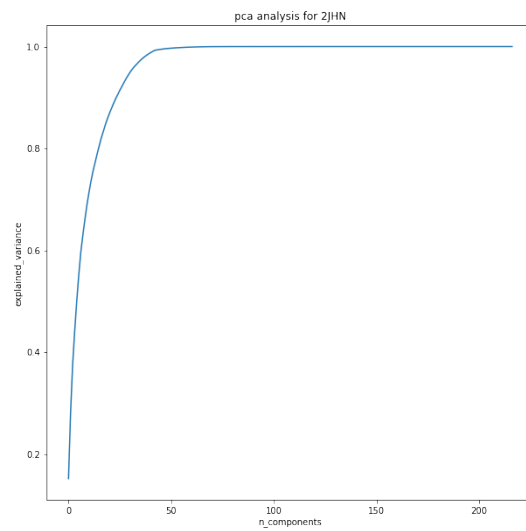
Following is the graph of cumulative explained variance ratio versus number of pca components for coupling type 3JHC

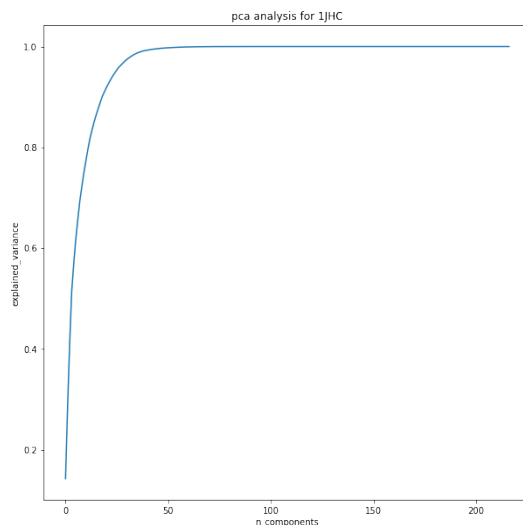


From this graph we clearly see that, keeping about 50 pca components for coupling type 3JHC already explains almost all the variance in the set of features used by us.

We can also do a similar analysis for other coupling types. The corresponding graphs are given below:







In each of these graphs we see that the cumulative explained variance ratio levels-off after about 50 pca components, hence keeping more pca components will have diminishing returns while increasing the memory cost of the feature set. Note that in our implementation, we wanted the computer to be able to automatically determine the no. of pca components to use. Since, it is a non-trivial exercise for the computer to identify where exactly do graphs level-off, therefore we settled for requiring the computer to chose pca components such that they explain 99.99% percent of the variance in the features space. This lead to the computer choosing somewhere between 75 to 78 pca component for each coupling type. In hindsight, we could have perhaps just set the number of pca component to be 50.

5.2 Reflection

Our approach towards solving the problem posed in this project can be summarized as follows:

1. EDA: A preliminary analysis of the data
2. Feature Engineering: Use the spatial coordinates of the atoms to extract their relative distance and spatial orientation
3. Preprocessing: Normalize the features and the target variable to have a unit variance and a zero mean by using scikit-learn's StandardScaler. Apply pca reduction to features.
4. Benchmarking: Train the benchmark model mentioned in section 2.4 to establish a baseline.
5. Modeling: For each type of scalar coupling, construct several different DNNs that attempt to the predict the scalar coupling constants. These models differ from each other in the number of their hidden layers. We also tried to build models using different regularization techniques such as 'BatchNormalization' vs 'Dropout', however, models with Dropout did not perform well and were ultimately discarded for the purposes of solving the current problem.
6. Ensembling: Having developed a couple of different models for each type, we too a weighted average of their predictions to obtain our final predictions. The validation set R_2 scores of each model were used as weights for their predictions, to obtain the averaged predictions.

7. Submission: We then use our models to predict the scalar couplings for the test set provided by CHAMPS and submitted these to Kaggle to obtain a final evaluation and score.

While, most of the steps were straight forward, there were 3 aspects of the final solution that we found specially interesting:

1. PCA reduction helps reduce over-fitting: This was most evident from the substantial improvement in the scores for coupling type 3JHC. When trained on the unreduced feature space, we obtained an R_2 score of 0.876 on the validation set as opposed to ~ 0.97 for the training set. With pca-reduction, the R_2 scores improved to 0.968 for the validation set as opposed to ~ 0.99 for the training set.
2. Scaling target values makes training better: If the range of target values of too large, then training can get adversely affect and become slow. Scaling the target values so as to have a unit variance helps improve this. This was unambiguously evident from the fact that our models took over 800 epochs to converge to their lowest errors when trained on unscaled target values. This reduced to about 200 epochs, without any significant loss of performance, after the target values were appropriately scaled.
3. Averaging over multiple different models gives better predictions: Though this is not surprising and has been repeatedly emphasized in machine learning literature, however, it was still quite satisfying to see this effect personally.

Another aspect of the project that we had not appreciated before this project was the need for writing a RAM efficient codes. Given the limited amount of RAM available on our laptop and on Google Colab, our code would frequently crash in the middle of an execution due to consuming too much RAM. We were therefore forced to find of more RAM efficient methods. We believe that these will help us in future also.

5.3 Improvement

Given the very high R_2 scores obtained during training, as well as the fact that inspite of using deeper and larger DNNs, our models did not produce vastly different mean absolute errors, makes us believe that this is the best that can be done with the current set of features. However, there are many other features that one can construct, such as angular orientations, bond hybridizations, molecular geometry and ring structures etc. As is evident from the discussion board on Kaggle, other participants in the competition have been able to successfully use the above features to provide better solutions to the problem. We therefore believe that including these features in our data will be the key to further improve our solution here.

References

- [1] CHAMPS-Kaggle, "Predicting Molecular Properties."
<https://www.kaggle.com/c/champs-scalar-coupling>.
- [2] P. E. Hansen, *Carbon—hydrogen spin—spin coupling constants*, *Progress in Nuclear Magnetic Resonance Spectroscopy* **14** (1981) 175 – 295.

- [3] H. J. Reich, "Spin-Spin Splitting: J-Coupling."
<https://www.chem.wisc.edu/areas/reich/nmr/05-hmr-03-jcoupl.htm/>.
- [4] Y. Rubin, "Coupling constants for ^1H and ^{13}C NMR."
<https://yvesrubin.files.wordpress.com/2011/03/coupling-constants-for-1h-and-13c-nmr.pdf>.
- [5] G. Montavon, K. Hansen, S. Fazli, M. Rupp, F. Biegler, A. Ziehe et al., *Learning invariant representations of molecules for atomization energy prediction*, 2012.