Problem Statement

A dataset in CSV format is provided. The dataset contains **100,000** samples. There are **304** input features, labeled **x001** to **x304**. The target variable is labeled as **y**.

The task is to create a model to predict the target variable y. Report the result of two evaluation metrics:

- Calculate and report the Root Mean Square Error (RMSE).
- Percentage accuracy of the model. Any predicted value with absolute error greater than 3.0 will be regarded as "wrong".

Comments

The data contains 100k samples and 304 features. This can be regarded as a large dataset. In large scale machine learning, usually mini-batch learning methods with stochastic gradient descent is applied. These could either linear methods such as regularized linear regression or SVM with SGD, or nonlinear estimators such as neural networks. Methods which apply SGD has several advantages and disadvantages. The advantages are,

- Efficiency. Mini-batch learning allows one to iteratively update the model using only a different portion of the training data at each iteration. This leads to efficiency in terms of computational load and memory usage, and allows to use huge datasets for optimization.
- Ease of implementation. Its straightforward to implement and debug the iterative routines, and debug the gradient calculations (gradient check)

Disadvantages are,

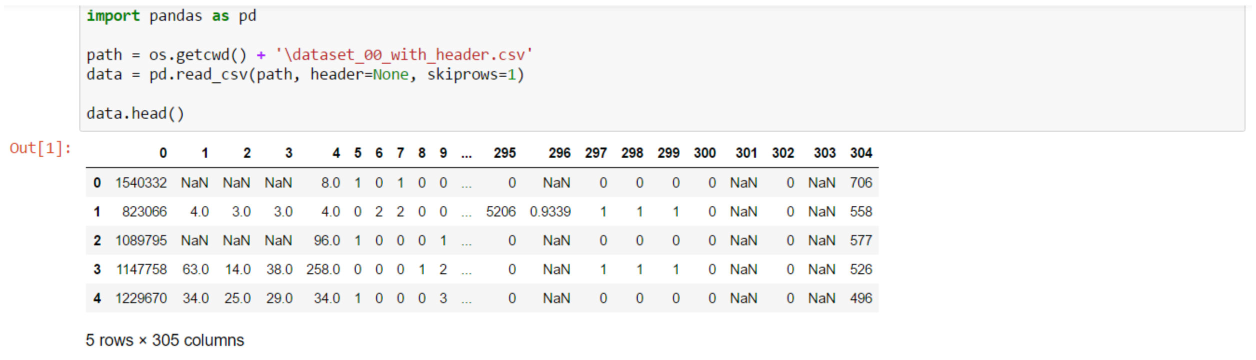
- Requires a number of hyper parameters to be optimized, such as the regularization and the number of iterations.
- Sensitive to feature scaling. The varying ranges of each feature leads to slow convergence and improper results due to the gradient update with a fixed learning rate.

There are also other kinds of algorithms based on decision trees and boosting, namely Random Forests and Gradient Boosting regressors. Decision trees are very effective especially in modeling categorical data, but the disadvantage is that they have computational problems with large-scale datasets.

I favor mini-batch SGD methods over batch methods. It is always better to start with a simpler model, may be a linear model and see if good results is obtained with your data. So I choose to start with **regularized linear regression with SGD** (named SGDRegressor in scikit-learn). I also tried to train a **regressor based on Random Forests** (RandomForestRegressor in scikit-learn). Due to the size of the dataset and my poor computational resources, I had serious problems to train a Random Forest Regressor. I was hoping to train a neural network, a non-linear SVM and a Gradient Boosting regressor, but I couldn't manage to find enough time and computational resource for those.

1. Importing the Data

The first step to take is to import the data and look through some statistics about the data and make some visualizations. This might help with understanding the data and give ideas about what kind of preprocessing steps might be helpful.

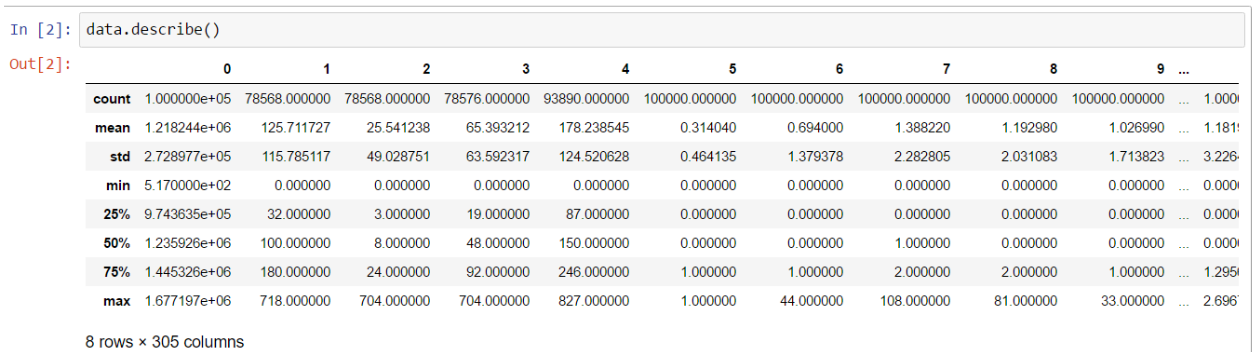


```
import pandas as pd

path = os.getcwd() + '\dataset_00_with_header.csv'
data = pd.read_csv(path, header=None, skiprows=1)

data.head()
```

Out[1]:

	0	1	2	3	4	5	6	7	8	9	...	295	296	297	298	299	300	301	302	303	304
0	1540332	NaN	NaN	NaN	8.0	1	0	1	0	0	...	0	NaN	0	0	0	0	NaN	0	NaN	706
1	823066	4.0	3.0	3.0	4.0	0	2	2	0	0	...	5206	0.9339	1	1	1	0	NaN	0	NaN	558
2	1089795	NaN	NaN	NaN	96.0	1	0	0	0	1	...	0	NaN	0	0	0	0	NaN	0	NaN	577
3	1147758	63.0	14.0	38.0	258.0	0	0	0	1	2	...	0	NaN	1	1	1	0	NaN	0	NaN	526
4	1229670	34.0	25.0	29.0	34.0	1	0	0	0	3	...	0	NaN	0	0	0	0	NaN	0	NaN	496

5 rows × 305 columns



In [2]:
```
data.describe()
```

Out[2]:

	0	1	2	3	4	5	6	7	8	9	...	
count	1.000000e+05	78568.000000	78568.000000	78576.000000	93890.000000	100000.000000	100000.000000	100000.000000	100000.000000	100000.000000	...	1.000
mean	1.218244e+06	125.711727	25.541238	65.393212	178.238545	0.314040	0.694000	1.388220	1.192980	1.026990	...	1.181
std	2.728977e+05	115.785117	49.028751	63.592317	124.520628	0.464135	1.379378	2.282805	2.031083	1.713823	...	3.226
min	5.170000e+02	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000
25%	9.743635e+05	32.000000	3.000000	19.000000	87.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000
50%	1.235926e+06	100.000000	8.000000	48.000000	150.000000	0.000000	0.000000	1.000000	0.000000	0.000000	...	0.000
75%	1.445326e+06	180.000000	24.000000	92.000000	246.000000	1.000000	1.000000	2.000000	2.000000	1.000000	...	1.295
max	1.677197e+06	718.000000	704.000000	704.000000	827.000000	1.000000	44.000000	108.000000	81.000000	33.000000	...	2.696

8 rows × 305 columns

It seemed there were a lot of missing values in the data, and first feature (column) had the greatest mean and variance.

2. Data Preprocessing

It would be better to take a **data imputation** step to fill the missing values. Other option is to drop the samples with missing values, which would result with losing a great number of samples. If we choose to drop samples with missing values, only 23 samples remain. So we choose to fill the missing values with mean value of the current feature column, however, other options such as filling with constant value/median/most_frequent/interpolation are available. This can be chosen in a cross-validation setting.

3. Splitting the data into train and test sets

The next important step is to split the data into training and test sets. This will allow us to test the models that we create on a different subset of the data then used to create the model. This allows us to avoid overfitting the model on the training set. We keep 20% portion of the data as test set, model selection will be performed on the remaining 80% portion.

```
# shuffle the data
data = data.sample(frac=1).reset_index(drop=True)

# lots of missing values in our data, fill out NaNs with 0!
#data.fillna(0, inplace=True)

# split the training data and leave 20% portion as test
X_train, X_test = train_test_split(data, test_size = 0.2)

cols = X_train.shape[1]
y_train = np.copy(X_train.iloc[:,-1])

# drop the label column from the train partition
X_train.drop(X_train.columns[cols-1], axis=1, inplace=True)
X_train.fillna(X_train.mean(), inplace=True)

y_test = np.copy(X_test.iloc[:,-1])

# drop the label column from the train partition
X_test.drop(X_test.columns[cols-1], axis=1, inplace=True)
X_test.fillna(X_train.mean(), inplace=True)
```

4. Feature Scaling

After data imputation, an important step is to perform **feature scaling**. It is almost always better to make features zero mean and unit variance, especially when applying gradient descent optimization. At each iteration, since the error is scaled by the feature itself, feature scale might lead to very large changes while updating the weight for that feature, and this might lead to slow convergence and improper results. Feature scaling can also be applied in a Pipeline setting as we will mention in the next section.

```
# normalize the data to zero mean unit variance
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train) # fit the scaler on training data
X_test_scaled = scaler.transform(X_test)
```

5. Hyper-Parameter Optimization

Machine learning models usually have many parameters to be optimized for the best performance. Cross-validation is the de-facto standard for this purpose. In *k*-fold cross-validation, training data is split into *k* folds. One of the folds are used as validation data, remaining *k*-1 folds are used as training data. Parameters are fit on the training data, and tested on the validation data. The process continued until each fold is once used as validation data. The mean cross-validation accuracy is computed, and the parameter set resulting with the best score is chosen. I have used *GridSearchCV* function of the *scikit-learn* library. *GridSearchCV* performs an exhaustive grid search in a cross-validation setting, to find the best parameters. I used 4-fold cv, so that each time 25% of the training data was used as validation data while 75% of it was used as training data.

5.1. SGD Regression Parameter Optimization

The parameters optimized for SGD regression was regularization weight *alpha*, type of the loss function, learning rate policy, regularization penalty function, and the epsilon. The *epsilon insensitive* loss function together with parameter *epsilon* corresponds to linear SVM loss function. I have fixed the number of iterations to 100, as the manual says that 10^6 iterations are sufficient for the SGD to converge in many applications. Since we have 60k training samples (20k was totally kept as test, and 20k as validation), 16.6 ~ 16 iterations suffice indeed.

```python
num_iter = 100
alpha = 10.0**-np.arange(1,7)

# our linear SGD regressor that we will optimize its regularization weight
model = SGDRegressor(n_iter=num_iter, random_state=0)

#param_grid = {"n_iter": num_iter, "alpha": alpha}
param_grid = {"alpha": 10.0**-np.arange(1,7),
             "loss" : ['squared_loss', 'epsilon_insensitive'],
              "learning_rate" : ['invscaling','optimal'],
              "penalty" : ['l2', 'l1', 'elasticnet'],
              "epsilon" : [0.05, 0.1, 0.15, 0.2]
             }

# Tune the hyperparameters by a grid search via 4-fold cross-validation
estimator = GridSearchCV(model,
                         param_grid,
                         cv=4,
                         n_jobs=1,
                        refit=True)

# start grid search to find the best set of parameters
estimator.fit(X_train_scaled[0:30000,:], y_train[0:30000])
```

I used the first 30k portion of the 80k training data in the cross-validation setting due to computational issues. The resulting best parameters and best estimator was found as,

```
{'penalty': 'l2', 'alpha': 0.001, 'learning_rate': 'optimal', 'loss':
'epsilon_insensitive', 'epsilon': 0.15}

SGDRegressor(alpha=0.001, average=False, epsilon=0.15, eta0=0.01,
fit_intercept=True, l1_ratio=0.15, learning_rate='optimal',
loss='epsilon_insensitive', n_iter=100, penalty='l2', power_t=0.25, random_state=0,
shuffle=True, verbose=0, warm_start=False)
```

After I obtained the best parameters using cross validation, I have cross validated the regularization weight using the best estimator on the full training data (80k portion), this time with a greater number of iterations, 3000. I have used the Pipeline property of sklearn, to combine the sgd regressor followed by standard scaler during cross validation, at each fold separately. The resulting best parameters and best estimator was found as,

```
{'sgd__alpha': 0.0001}

Pipeline(steps=[('norm', StandardScaler(copy=True, with_mean=True, with_std=True)),
('sgd', SGDRegressor(alpha=0.0001, average=False, epsilon=0.15, eta0=0.01,
       fit_intercept=True, l1_ratio=0.15, learning_rate='optimal',
       loss='epsilon_insensitive', n_iter=3000, penalty='l2', power_t=0.25,
       random_state=0, shuffle=True, verbose=0, warm_start=False))])
```

```
num_iter = 3000
alpha = 10.0**-np.arange(1,7)

# our linear SGD regressor that we will optimize its regularization weight
model = SGDRegressor(average=False, epsilon=0.15, eta0=0.01,
       fit_intercept=True, l1_ratio=0.15, learning_rate='optimal',
       loss='epsilon_insensitive', n_iter=100, penalty='l2', power_t=0.25,
       random_state=0, shuffle=True, verbose=0, warm_start=False)

pipe = Pipeline(steps=[('norm', StandardScaler()), ('sgd', model)])

parameters = {'sgd__alpha': alpha}

# Tune the hyperparameters by a grid search via 4-fold cross-validation
estimator = GridSearchCV(estimator=pipe,
                         param_grid=parameters,
                         cv=4,
                         scoring='mean_squared_error',
                         n_jobs=-1,
                         refit=True)

# start grid search to find the best set of parameters
estimator.fit(X_train, y_train)
```

5.2. Random Forest Regression Parameter Optimization

Random Forest regression do not include an SGD optimization. It is a batch learning method in that it uses all the training data to train a model at once. To build a forest, it calculates the best combination of many decision trees within a bootstrapped setting. Each tree performs an implicit feature selection process, to selects the best feature to split the training data. So we end up with a relative feature importance when the training is finished. This could be useful to decide discarding the least useful features in advance.

The parameters that I optimized were,

- *n_estimators*: The number of trees in the forest.
- *max_depth*: The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than *min_samples_split* samples.
- *max_features*: The number of features to consider when looking for the best split.
- *min_samples_split*: The minimum number of samples required to split an internal node.
- *min_samples_leaf*: The minimum number of samples required to be at a leaf node.
- *bootstrap*: Whether bootstrap samples are used when building trees.

The resulting best parameters and best estimator was found as follows,

```
Pipeline(steps=[('norm', StandardScaler(copy=True, with_mean=True, with_std=True)), (
'rf', RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_split=1e-07, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=1000, n_jobs=1, oob_score=False, random_state=0,
           verbose=0, warm_start=False))])
```

```
## HyperParameter Optimization ##
model = RandomForestRegressor(random_state=0)

param_grid = {"rf__n_estimators": [500, 800, 1000],
    "rf__max_depth": [8, None],
    "rf__max_features": ['sqrt', 'log2', None],
    "rf__min_samples_split": [2, 5, 10],
    "rf__min_samples_leaf": [1, 3, 10],
    "rf__bootstrap": [True, False]}

pipe = Pipeline(steps=[('norm', StandardScaler()), ('rf', model)])

# Tune the hyperparameters by a grid search via 4-fold cross-validation
estimator = GridSearchCV(estimator=pipe,
                         param_grid=param_grid,
                         cv=4,
                         scoring='mean_squared_error',
                         n_jobs=-1,
                         refit=True)


# start grid search to find the best set of parameters
estimator.fit(X_train, y_train)
```

6. Training and Prediction

With the *refit=True* option, *GridSearchCV* trains the best model with the available training data (training + validation). This gives the model *best_estimator_*. The best estimator is the model chosen based on the best scores obtained with the validation data. The best estimator thus could be used to make predictions using the Training data and Test data. As a final step I have trained my best estimator using all data available (100k) and performed prediction with the trained model.

7. Results

The error obtained with the Training data corresponds to **training error**, whereas error obtained with the Test data corresponds to **test error**, or **generalization error**. Overall, among the trained models, it is always better to choose the one with the best generalization error. After calculating and reporting the training and test error, I have performed a final training and prediction on the whole data. All of the results for SGD Regression are given as,

SGDRegressor RMSE X_train_scaled:	47.38
SGDRegressor RMSE X_test:	47.90
SGDRegressor RMSE Final:	47.43
SGDRegressor Prediction Accuracy Final:	6.56

It seems that our sgd model performed very poorly. We have a **bias problem** with our sgd regression model, because both our training and test errors are **high**. We are **underfitting** our data. We could even,

- use a more complex nonlinear model such as neural networks or SVM with nonlinear kernel, or,
- we don't change our model but even try
    - getting additional features
    - adding polynomial features
    - decreasing regularization weight *alpha*

```
y_true_train, y_pred_train = y_train, estimator.best_estimator_.predict(X_train)

# print out the training RMSE
err_train = mean_squared_error(y_true_train, y_pred_train)
err_train = err_train**0.5
print("SGDRegressor RMSE X_train: %.2f" % err_train)


# print out the test RMSE
y_true, y_pred = y_test, estimator.best_estimator_.predict(X_test)
err_test = mean_squared_error(y_true, y_pred)
err_test = err_test**0.5
print("SGDRegressor RMSE X_test: %.2f" % err_test)
```

It might also be helpful to perform a dimensionality reduction such as PCA first, to come up with the best features capturing the greatest amount of variance in the data. This could also help to prevent the redundancy with the current features.

It is also important to note that **Learning Curves** are important to analyze the strengths and weaknesses of a trained model. The learning curve tells much about the issues with a model, in a varying number of samples setting. It tells whether there is a bias or variance problem or not, whether the model is underfitting or overfitting the data.

Note: Training of my Random Forest Regression model took more than 36 hrs. Average utilization of CPU was about 90% and memory usage was about 2GB. **The resultant regression model size was 6.55 GB, so I couldn’t add it with my submission**. Below are the results of my best random forest model estimated with cross validation.

RandomForestRegressor RMSE X_train:	10.41
RandomForestRegressor RMSE X_test:	27.97
RandomForestRegressor RMSE Final:	10.26
RandomForestRegressor Prediction Accuracy Final:	36.89

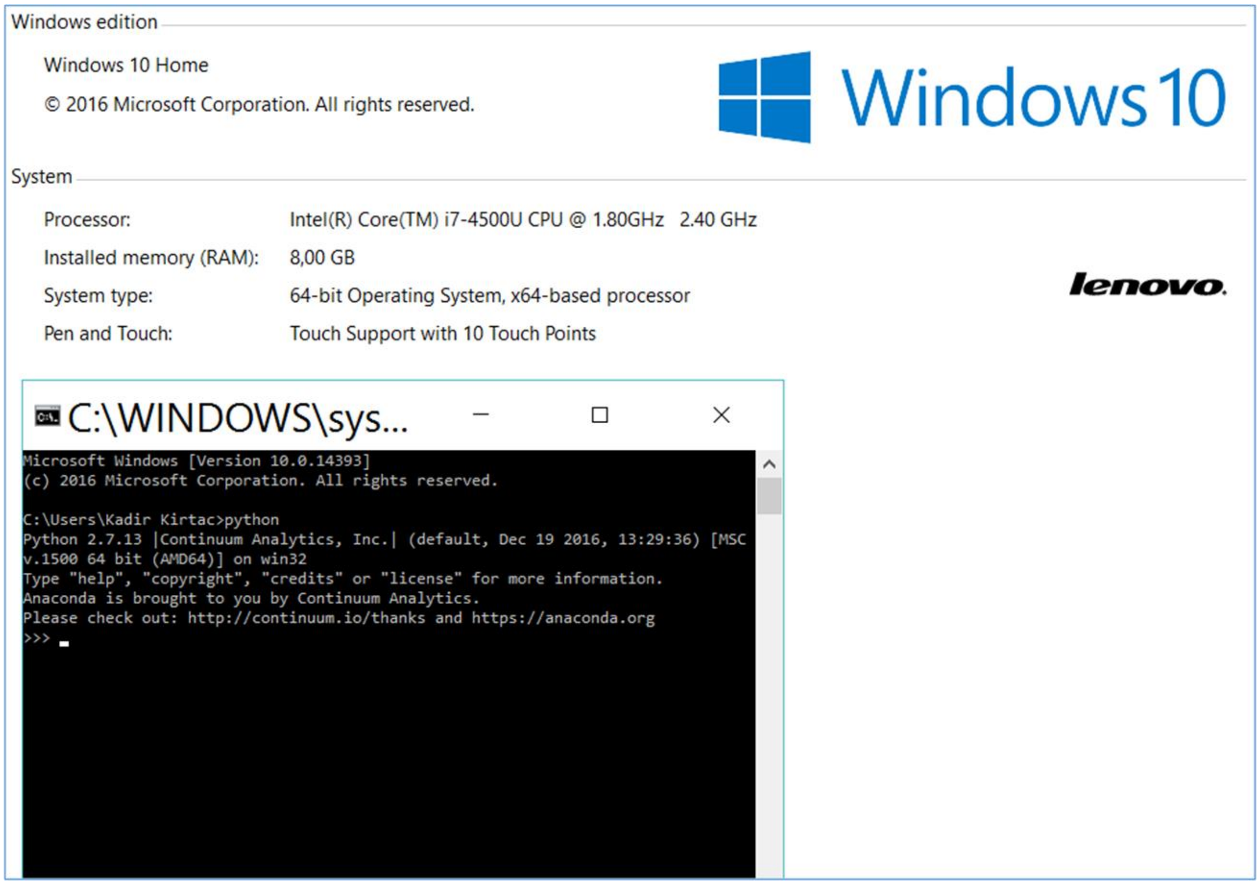
Results of random forest regression are more promising than the results with sgd regression. Training and test errors have good separation, however, results have a lot to improve. Collecting more data or adding features might help to reduce training error. To reduce the computational burden of training, an initial feature selection step might help to eliminate redundant features. Dimensionality reduction methods such as PCA might also help to obtain the most promising features. I wasn’t able to plot feature importance values given by the trained model due to computational issues. Feature importance values might help to eliminate redundant features.

8. Instruction Manual

I have developed the code using Python 2.7.13 shipped with MiniConda, on Windows 10. My system properties and development environment is summarized in Figure 1.

Package dependencies are as follows,

- numpy 1.12.1
- scikit-learn 0.18.1
- pandas 0.19.2



```
Windows edition
    Windows 10 Home
    © 2016 Microsoft Corporation. All rights reserved.

System
    Processor:                Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz   2.40 GHz
    Installed memory (RAM):   8,00 GB
    System type:              64-bit Operating System, x64-based processor
    Pen and Touch:            Touch Support with 10 Touch Points
```

```
C:\WINDOWS\sys...
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Kadir Kirtac>python
Python 2.7.13 |Continuum Analytics, Inc.| (default, Dec 19 2016, 13:29:36) [MSC
v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>> _
```

Figure 1 Development Environment

8.1. Usage

I have zipped all my work in a folder named "**Kadir-WalletHub**". The folder contains all training and resultant model files. Training files are named as **train-linear-SGD.py** and **train-random-forest.py**. These files read the training file hard-coded and assumes that its name is **dataset_00_with_header.csv,** and it is in the same folder with the training files. There is also the prediction file "**make_prediction.py**" that you will use to make predictions with your data. This code reads and uses a prediction model, namely **regressor_sgd.pkl**. The code **must** be in the same folder with the model file. You can call the code in the command line via "**python make_prediction.py path_to_your_dataset**". You can give an additional file path parameter after the dataset path, to make the predicted values saved to that file. Otherwise, predicted values are saved to a text file named "result.out" in the same folder with the code. I have assumed in the code that your dataset is in the same format with the training data, which is a CSV file containing the label in the last column.