Assignment 3: Data mining in action

Piyush Vats

STUDENT# 13046899

The Problem, Inputs & Outputs 3
Inputs & Outputs 3
Problem Exploration 4
Data Pre-processing 4
The use of KNIME was discontinued and python used from hereafter. 5
Exploring Classifiers 6
Tuning Classifier 7
Conclusion 11
Bibliography 11

The Problem, Inputs & Outputs

Working in the capacity of a data scientist, the task in hand is to explore the dataset provided and make appropriate classifications for a property in question.

Specifically, data regarding property characteristics was available and the task at hand was to build a classifier to predict the Qualified status of the property in question where the qualified attribute has a value of either {0(U) or 1(Q)}.

Inputs & Outputs

Training set was provided via filename: TrainingData.csv where each entity (property in this case) had already been qualified. Attribute names and random sample of the raw data are highlighted below.

```
Reading data

Out[1]: Index(['SSL', 'BATHRM', 'HF_BATHRM', 'HEAT', 'HEAT_D', 'AC', 'NUM_UNITS',
       'ROOMS', 'BEDRM', 'AYB', 'YR_RMDL', 'EYB', 'STORIES', 'SALEDATE',
       'PRICE', 'QUALIFIED', 'SALE_NUM', 'GBA', 'BLDG_NUM', 'STYLE', 'STYLE_D',
       'STRUCT', 'STRUCT_D', 'GRADE', 'GRADE_D', 'CNDTN', 'CNDTN_D', 'EXTWALL',
       'EXTWALL_D', 'ROOF', 'ROOF_D', 'INTWALL', 'INTWALL_D', 'KITCHENS',
       'FIREPLACES', 'USECODE', 'LANDAREA', 'GIS_LAST_MOD_DTTM',
       'SALEDATE_Year', 'SALEDATE_Quarter', 'SALEDATE_Month'],
      dtype='object')
```

Figure 1: Attribute Names

```
training_table.head

Reading data

Out[100]: <bound method NDFrame.head of                SSL  BATHRM  HF_BATHRM  HEAT         HEAT_D AC  NUM_UNITS  \
row ID
28915   1605    0050       2          0    13  Hot Water Rad  Y          1
36804   2256    0831       3          1     1     Forced Air  Y          1
11913   0938    0021       2          1     1     Forced Air  Y          1
41326   2588    0025       2          1     7      Warm Cool  Y          2
88621   5154    0906       1          0    13  Hot Water Rad  N          1
31340   1957    0069       1          2    13  Hot Water Rad  Y          1
44747   2751    0005       4          1     7      Warm Cool  Y          1
14948   1011    0061       1          1     7      Warm Cool  Y          2
46015   2774    0020       2          1    13  Hot Water Rad  N          1
12430   0996    0024       1          1     1     Forced Air  Y          1
13128   0908    0061       1          0    13  Hot Water Rad  N          1
57883   3165    0812       1          0     3   Wall Furnace  N          1
10749   0946    0010       3          0    13  Hot Water Rad  Y          3
102158  5777    0817       1          0     1     Forced Air  Y          1
105562  6118    0028       4          0    13  Hot Water Rad  N          4
106336  PAR 02060063       2          2    13  Hot Water Rad  Y          1
```

Figure 2: Sample data which includes first few attributes

Also provided was a second set of data via filename: TestingSet.csv, similar to the training set without qualified status. The task at hand was to create an appropriate classifier that classified each property based on its attributes. Expected output was to include two columns, “rowid” and “QUALIFIED” which map to the rowed of equivalent property and the predicted qualified value.

Problem Exploration

Data Pre-processing

Step 1

Data is explored using the File Reader node. SALEDATE attribute type is set to Zone date time so that it can be processed further later. It is noticed that several attributes have missing values that need to be replaced appropriately.

Step 2

The SALEDATE attribute is processed through multiple KNIME nodes to split date up into YEAR, QUARTER and MONTH and stored in separate columns. Node setup is highlighted below.

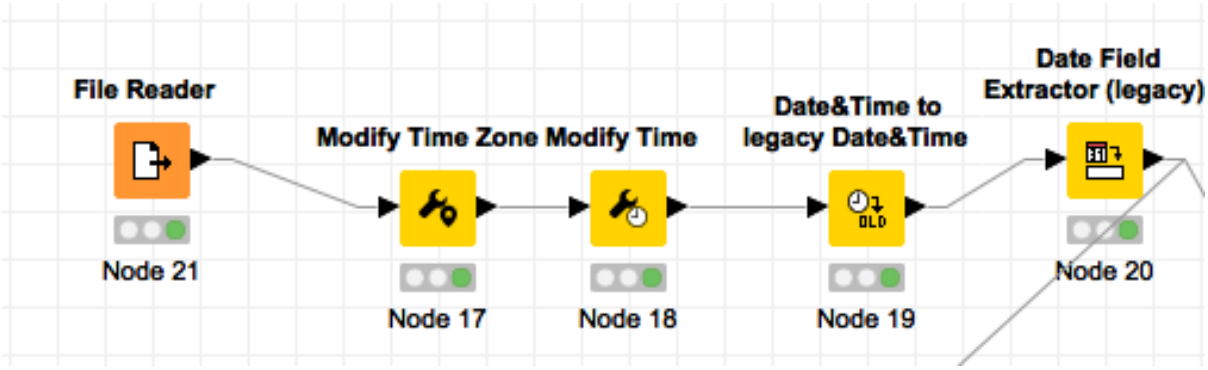

Figure 3: Converting string SALEDATE to Year, Quarter and Month.

Step 3

Attributes deemed to not be necessary for accurate classification were filtered out. Specifically, categorical attributes such as HEAT_D, STYLE_D etc. already had assigned numerical values. As such, these attributes were removed using the Column Filter Node. A summary of which attributes were removed is highlighted below.

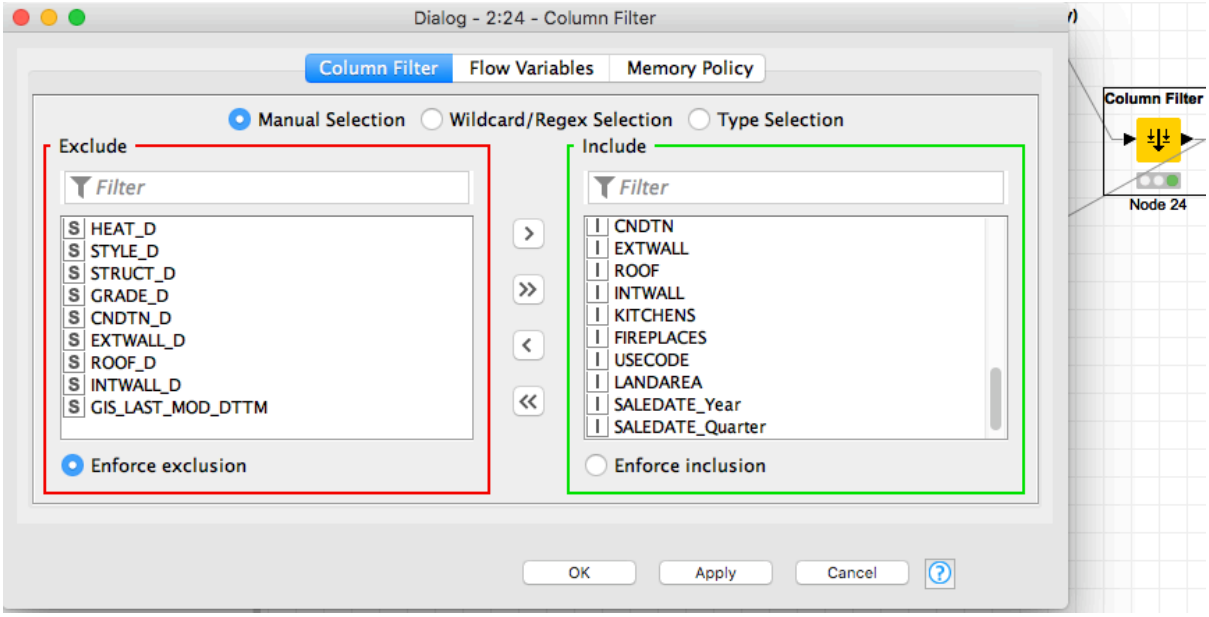

Figure 4: Filtering redundant attributes

Step 4

Missing values need to be replaced as this is needed to run sickie-learn classifiers without errors. Several approaches were tried (mean, median, zero, outlier). After manual observation of raw data with missing attributes and running baseline performance testing of several classifiers under different replacement methods, it was concluded that missing numerical data would be replaced by an outlier value as this provided the highest accuracy rates across several classifier methods. Missing Value node in KNIME was used to do the needed (Figure 5)

Step 5

"PRICE" attribute was normalised via MIN-MAX normalisation. (Figure 5)

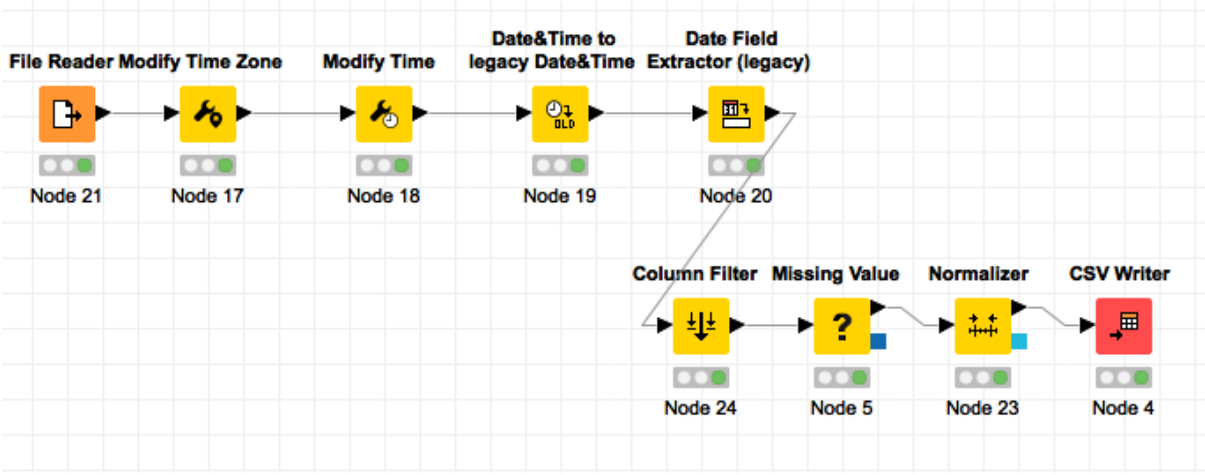

Figure 5: KNIME data pre-processing via KNIME

Step 6

The use of KNIME was discontinued and python used from hereafter.

We need to convert the "AC" attribute from string to Boolean value, where 1 means YES and 0 means NO, missing values were replaced by outlier value of -10. Pandas was used to read pre-processed data via knime into the python work environment and convert "AC" field into a Boolean as highlighted in Figure 6.

```
In [47]: training_table = pd.read_csv("trainingd.csv", header=0, index_col=0)

In [48]: ACYES_mask = training_table['AC'] == 'Y'
         ACNO_mask = training_table['AC'] == 'N'
         ACNO_null = training_table['AC'] == '-10'
         training_table.loc[ACYES_mask, 'AC_Boolean'] = 1
         training_table.loc[ACNO_mask, 'AC_Boolean'] = 0
         training_table.loc[ACNO_null, 'AC_Boolean'] = -10
```

Figure 6: Binarise AC column

Step 7

Pre-processed training data is now split into feature and label sets via the code highlighted below in Figure 7

```
In [ ]: train_labels = np.array(iris_table['QUALIFIED'])
        iris_data = iris_table.loc[:,['BATHRM','HF_BATHRM','HEAT','AC_Boolean','NUM_UNITS','ROOMS','BEDRM',
                                      'AYB','EYB','STORIES','PRICE','SALE_NUM','GBA','BLDG_NUM','STYLE','STRUCT'
                                      ,'GRADE','CNDTN','EXTWALL', 'ROOF','INTWALL','KITCHENS','FIREPLACES','USECODE',
                                      'LANDAREA', 'SALEDATE_Year', 'SALEDATE_Quarter', 'SALEDATE_Month']]
        feature_list = list(iris_data.columns)
        train_features = np.array(iris_data)
```

Figure 7: Splitting pre-processed data into attributes and label.

Exploring Classifiers

After exploring the dataset, it is noted that it is a classification problem, all attributes are labelled. Grid Search library is leveraged via sickie-learn to test several shortlisted classifier models and a combination of their attributes to establish baseline accuracy. Once this is done, best performers will be selected and attributes tuned further to try and achieve better accuracy. Grids each based hyper parameter optimisation across multiple models setup (Batista, 2018) was used to achieve the above. Python code used can be found on the source website and reviewed during the oral presentation.

The classifier models tested are as follows:

- ExtraTreeClassifier
- RandomForestClassifier
- SVC
- RandomForestClassifer
- DecisionTreeClassifier

It is to be noted that the aim of this process was to establish base performance. It was clear from the evidence that the Random forest classifier and decision tree classifier were the best performing as the top 5 results were all attributed to them albeit with varying attribute settings. They are shown below:

Out[111]:

	estimator	min_score	mean_score	max_score	std_score	criterion	max_depth	n_estimators
13	RandomForestClassifier	0.902027	0.903047	0.903826	0.000754137	NaN	NaN	32
15	RandomForestClassifier	0.89634	0.898012	0.899544	0.00131168	gini	NaN	16
12	RandomForestClassifier	0.895726	0.897513	0.898939	0.00133638	NaN	NaN	16
9	DecisionTreeClassifier	0.896327	0.897377	0.898838	0.00106587	entropy	9	NaN
3	DecisionTreeClassifier	0.895982	0.897199	0.89898	0.00128693	gini	7	NaN

Figure 8: Grid search Summary

As part of the task, I was assigned the decision tree classifier and seeing how the difference in accuracy between random forest classifier and decision tree classifier was negligible, chose this as the classifier to try and tune further and achieve higher accuracy. This time around grid search was used to further explore the combination of parameters that would achieve the highest accuracy.

Tuning Classifier

The approach to choosing the best classifier was based truly on performance achieved through extensive search of a different combinations of parameter values for each classifier.

It is to be noted that my data pre-processing capabilities were quite limited due to lack of experience and in hindsight, it makes sense that the decision tree performed well as it

- can handle both numerical and nominal data. This is the case with the training data used in this case
- is capable of handling a high degree of missing/erroneous data as is the case with the training data used in this case
- is an "eager learner" in that it needs to be trained using a training set and then classifies new data. Suitable as we were given a training set with known labels and a testing set to classify thereafter.
- can create a classifier without the need for extensive data pre-processing. Ideal as pre-processing skills and experience is limited.
- Auto picks the best features from tabulated data instead of having to manually do so.

As is evident, all of the above conditions are prevalent in the problem scenario being explored in this task. Keeping this in mind along with the fact that my experience with identifying correlations between raw data and given labels etc. is limited, the limitations placed in building models such as Naive Baines and KNN where such observational skills are critical to the classifier's accuracy, decision tree is the best option.

Grid Search was again used to explore the plethora of parameters and the best combination identified to achieve the highest accuracy possible.

The 4 most critical parameters for tuning of the Decision Tree Classifier were identified as (Fraj, 2017)

- max_depth – depth of tree: The deeper the tree, the more attributes it captures in its decision making process. Depth of 1 to 32 was explored as shown in in Figure 9. It to be noted that depth between 5-10 seems to be appropriate with larger depths resulting in overfitting.
- min_sample_split – minimum number of samples to split node. Tested values between 10 -100%. As shown in in Figure 10, value of 20% and above seems to be optimal and anything below seems to result in under fitting.

- min_sample_leaf – Minimum number of samples needed to be at a leaf node. As can be seen in Figure 11, increasing min_sample_leaf to more than .40 will result in under fitting of data.
- max_features – Number of features to take into account in the decision to make the best split. In this scenario, number of features for our case is queried and iterated. As can be noted in figure 12, anything below 9 seems to result in data under fitting with results being constant for values of 9 and above.

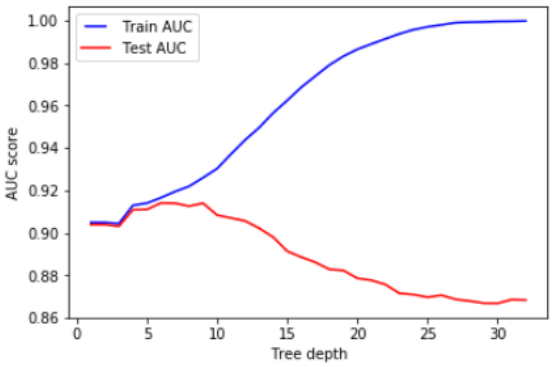

Figure 9: Tree Depth

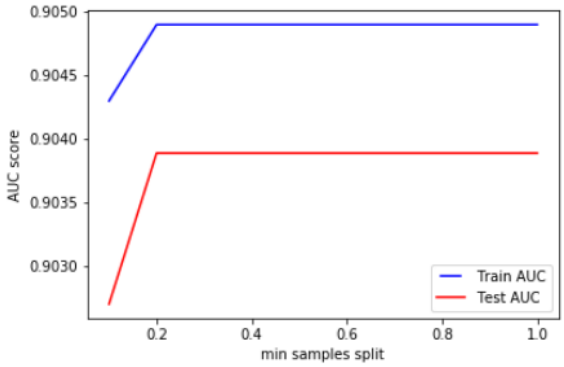

Figure 10: Min sample size

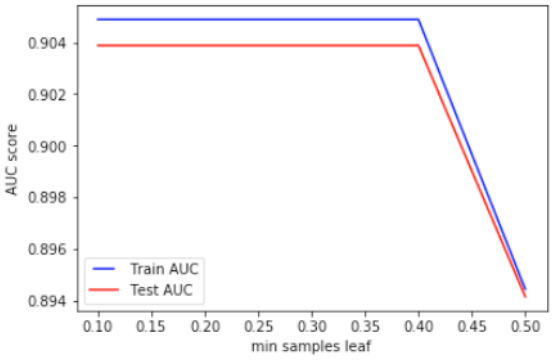

Figure 11: Min sample leaf

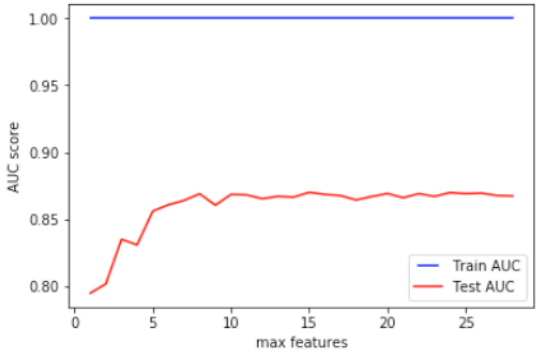

Figure 12: max features

```
In [264]: dt = tree.DecisionTreeClassifier()
          dt.fit(X_train, y_train)
          y_pred = dt.predict(X_test)
          alse_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
          roc_auc = auc(false_positive_rate, true_positive_rate)
          roc_auc

Out[264]: 0.8990062056820005
```

Figure 13: Accuracy of Decision Tree Classifier without tuning

```
In [299]: dt = tree.DecisionTreeClassifier(max_depth = 9,max_features = 10, min_samples_leaf=.1,min_samples_split=.5)
          dt.fit(X_train, y_train)
          y_pred = dt.predict(X_test)
          alse_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
          roc_auc = auc(false_positive_rate, true_positive_rate)
          roc_auc

Out[299]: 0.9466728037694071
```

Figure 14: Accuracy of Decision Tree Classifier post tuning.

As can be concluded from Figure 13 and 14, post tuning, a performance increase of approximately 5.3% was achieved.

As is necessary to explore all possible techniques when exploring a classification problem such as this, an effort was made to use ensemble techniques such as voting, bagging and gradient boosting were explored and optimised using grid search and alternative optimising techniques (Figure 15, 16, 17)

Theoretically, the Voting Classifier would allow us to utilise multiple basic classifier models and combine the results to reduce overall error.

The Gradient Boosting Classifier is a type of boosting where the model relies on the intuition that the best possible next model when combined with previous models, minimizes the overall prediction error. (Unknown, Na).

Finally, Bagging or bootstrap aggregation is used to reduce variance for classifiers with high variance, such as the decision tree classifier chosen by us.

Although theoretically all of the above classifiers should have improved on the performance marker of the tuned Decision Tree Classifier, performance increase was not achieved over a tuned Decision Tree Classifier. Admittedly, given more time, experience knowledge of how these classifiers can be optimised, a performance increase is possible

```
In [237]: learning_rates = [0.05, 0.1, 0.25, 0.5, 0.75, 1]
          for learning_rate in learning_rates:
              gb = GradientBoostingClassifier(n_estimators=100, learning_rate = learning_rate, max_features=10, ma
              gb.fit(train_features, train_labels)
              print("Learning rate: ", learning_rate)
              print("Accuracy score (training): {0:.3f}".format(gb.score(train_features, train_labels)))
              print("Accuracy score (validation): {0:.3f}".format(gb.score(test_features, test_labels)))
              print()

          Learning rate:  0.05
          Accuracy score (training): 0.924
          Accuracy score (validation): 0.912

          Learning rate:  0.1
          Accuracy score (training): 0.935
          Accuracy score (validation): 0.916

          Learning rate:  0.25
          Accuracy score (training): 0.958
          Accuracy score (validation): 0.913

          Learning rate:  0.5
          Accuracy score (training): 0.982
          Accuracy score (validation): 0.906

          Learning rate:  0.75
          Accuracy score (training): 0.989
          Accuracy score (validation): 0.898

          Learning rate:  1
          Accuracy score (training): 0.993
          Accuracy score (validation): 0.894
```

Figure 15: Gradient Boosting Classifier test

```
In [304]: from sklearn.cross_validation import cross_val_score
          from sklearn.ensemble import BaggingClassifier

In [305]: bagging = BaggingClassifier(dtc, max_samples=0.4, max_features=.5, n_estimators=60)

In [306]: predit1 = bagging.fit(train_features, train_labels)

In [307]: bagging.score(test_features, test_labels)

Out[307]: 0.8927047781569966
```

Figure 16: Bagging Classifier test

```
In [308]: from sklearn.tree import DecisionTreeClassifier
          from sklearn.ensemble import GradientBoostingClassifier

          regr_1 = DecisionTreeClassifier(criterion='gini',
                      max_depth=7)

          regr_2 = GradientBoostingClassifier(n_estimators=500, learning_rate = 0.1, max_features=2, r

          regr_1.fit(train_features,train_labels)
          regr_2.fit(train_features, train_labels)

          y_1 = regr_1.predict(test_features)
          y_2 = regr_2.predict(test_features)

          from sklearn.ensemble import VotingClassifier

          eclf = VotingClassifier(estimators=[('dt', regr_1),('gra', regr_2)], voting='soft')

          eclf = eclf.fit(train_features, train_labels)


          pred_eclfa = eclf.predict(test_features)
          eclf.score(test_features, test_labels)

Out[308]: 0.9123826791808873
```

Figure 17: Voting Classifier test

Conclusion

After exploring multiple classifiers, it was decided upon based on performance markers, that the decision tree classifier was the most efficient at classifying properties. As highlighted in the report previously, this specific classifier suited the problem at had because it

- can handle both numerical and nominal data, perfect for the training data provided for this classifier problem
- is capable of handling a high degree of missing/erroneous data as is the case with the training data used in this case
- is an “eager learner” in that it needs to be trained using a training set and then classifies new data. Suitable as we were given a training set with known labels and a testing set to classify thereafter.
- can create a classifier without the need for extensive data pre-processing. Ideal as pre-processing skills and experience is limited.
- Auto picks the best features from tabulated data instead of having to manually do so.

The chosen classifier was then optimised/tuned using the grid search library provide by the sci-kit learn ML library. The attributes and their values (Figure 18) were chosen as the combination of them provided an approximately 5% improvement in performance when compared to the default classifier with an accuracy of 94.7%.

```
In [299]: dt = tree.DecisionTreeClassifier(max_depth = 9,max_features = 10, min_samples_leaf=.1,min_samples_split=.5)
          dt.fit(X_train, y_train)
          y_pred = dt.predict(X_test)
          alse_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
          roc_auc = auc(false_positive_rate, true_positive_rate)
          roc_auc

Out[299]: 0.9466728037694071
```

Bibliography

Gradient Boosting – The Coolest Kid on The Machine Learning Block [Online] / auth. Unknown // DISPLAYR. - Na Na, Na. - 10 05, 2018. - https://www.displayr.com/gradient-boosting-the-coolest-kid-on-the-machine-learning-block/.

Hyperparameter optimization across multiple models in scikit-learn [Online] / auth. Batista David // www.davebatista.com. - 02 23, 2018. - 10 04, 2018. - http://www.davidsbatista.net/blog/2018/02/23/model_optimization/.

InDepth: Parameter tuning for Decision Tree [Online] / auth. Fraj Mohtadi Ben // Medium.com. - 12 21, 2017. - 10 5, 2018. - https://medium.com/@mohtedibf/indepth-parameter-tuning-for-decision-tree-6753118a03c3.