Classification of unlabeled LoL match records with "Win/Loss"

■Introduction:

Cliffhanger is really important to a game. Creators of League of Legends need to close the gap between teams during the game to improve the fun of the game. To achieve this, they should predict solo players' combat gains. This project aims to make use of big data to optimize different models which could classify the results of the game according to various fields of game's information. The fields contain: Game ID, Creation Time (in Epoch format), Game Duration (in seconds), Season ID Winner (1 = team1, 2 = team2), First Baron, dragon, tower, blood, inhibitor and Rift Herald (1 = team1, 2 = team2, 0 = none), the number of towers, inhibitor, Baron, dragon and Rift Herald kills each team has.

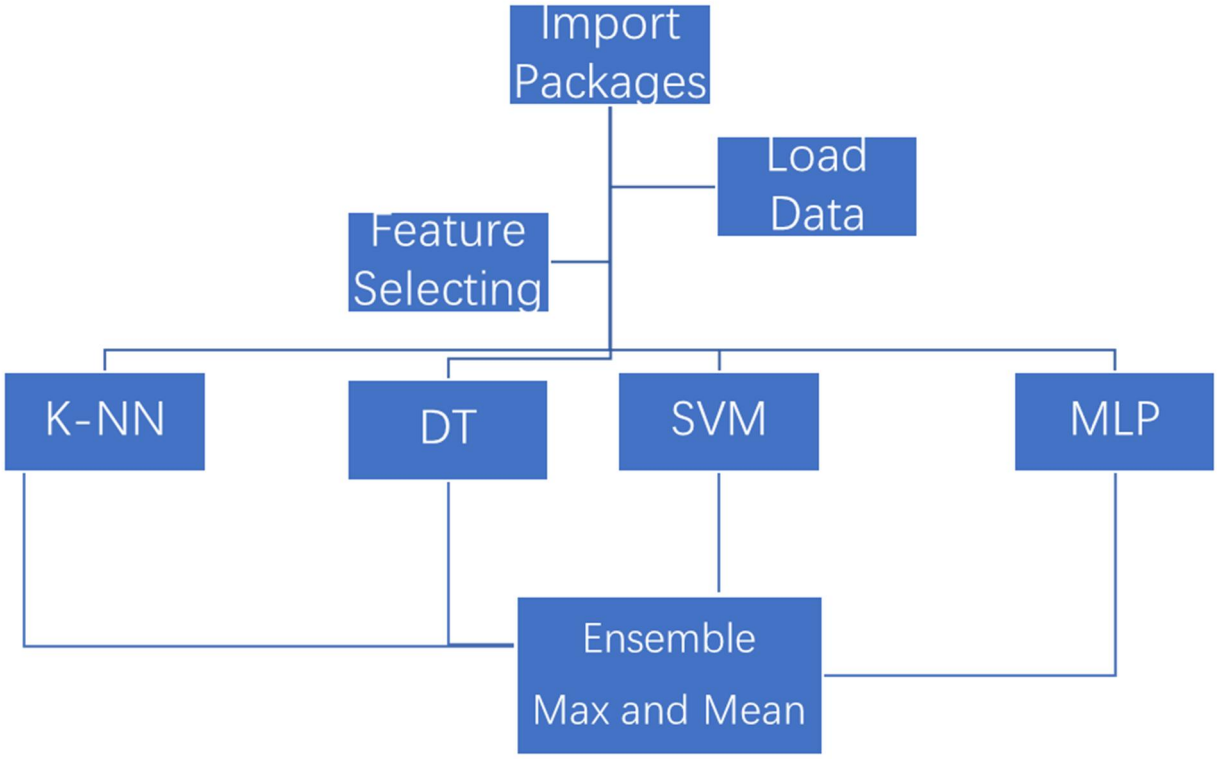
As for the requirements of the project, it should at least work out one classifier, whose accuracy should be higher than 50%. What's more, it is better to optimize several kinds of models and make comparison based on their accuracy and training time. The comparison will help us develop a further comprehension and ideas to make the classifiers more effective. Parameters of each model are supposed to be printed out, too.

The project includes two programs: classifiers.py and TreeDrawing.py. The first one consists of several kinds of models and the second one is to visualize the Decision Tree for a better understanding.

The program classifiers.py includes four classifiers: K-NN, Decision Tree, Support Vector Machine and MLP. To make the project more comprehensive, two voting ensemble models whose fusion methods are continued-valued, max and mean, are added. These models are trained by the training dataset 'new_data.csv' and are put into use for the test dataset 'test_set.csv'.

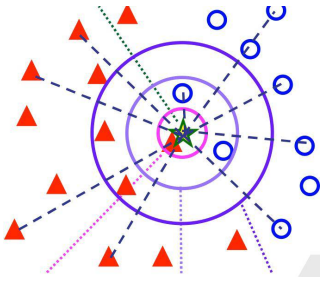
■ Algorithms:

1. Flow chart:



2. **Preparation:** First of all, the program does the data preparation to load the data. Then it selects the features and labels. The first four attributes (GameId, CreationTime, GameDuration, SeasonId) are taken away because they contribute little to the result of the game.

3. **K-NN:** K-Nearest Neighbor is a simple classifier. The initial state of K-NN classification is the sample of some known categories (represented by all eigenvectors). For a new sample of an unknown category, we find k neighbor samples with the smallest distance, and we assume that the new sample also belongs to that class based on which of these k neighbors belong to the most.



In this project, after adjustments, the parameters of this model are:

```
K-NN :
parameters:  {'algorithm': 'auto', 'leaf_size': 30, 'metric': 'minkowski', 'metric_params':
None, 'n_jobs': None, 'n_neighbors': 5, 'p': 2, 'weights': 'uniform'}
```

a. **n_neighbors** stands for k. Now k is 5. The new data will follow the label which is the most among five closest data from new data.

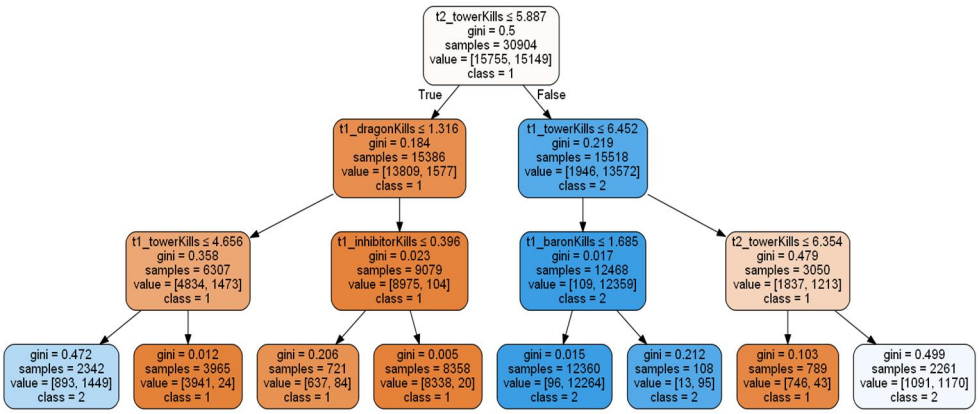
b. **p**=2 means the algorithm uses Euclidean distance.

c. **Leaf_size** is the default value 30. This value controls the threshold for the number of leaf nodes to stop building subtrees using kd or spherical trees. The smaller this value is, the larger the generated KC tree or ball tree will be, the deeper the number of layers will be, and the longer the building time will be.

d. **Algorithm**: Algorithm with finite radius nearest neighbor, which can be 'auto', 'ball_tree', 'kd_tree', 'Brute'. 'Brute' corresponds to the first linear scan; 'KD_tree' corresponds to the second kd tree implementation; 'ball_tree' corresponds to the realization of the third kind of ball tree; 'Auto' will make tradeoff among the above three algorithms and choose the optimal algorithm that fits the best.

e. **Weights**: {'uniform', 'distance'}, default='uniform'. The program uses 'uniform' because it is more accurate.

4. **Decision Tree**: Decision Tree is one of the most widely used and practical methods for inductive inference. A Decision Tree is a tree structure (which can be binary or non-binary). Each of its non-leaf nodes represents a test on a feature property, each branch represents the output of the feature property on a range, and each leaf holds a category. The process of using Decision Tree to make decision is to start from the root node, test the corresponding characteristic attributes in the item to be classified, and select the output branch according to its value until it reaches the leaf node, and take the category stored by the leaf node as the decision result.



In this project, the parameters of the Decision Tree are:

```
Decision Tree :
parameters:  {'class_weight': None, 'criterion': 'entropy', 'max_depth': 15, 'max_features': None,
'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1,
'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'presort': False, 'random_state': None,
'splitter': 'random'}
accuracy:  0.9615272515301662
training time:  0.025931358337402344
```

a. **Criterion**: A measure of the quality of the partition. Gini or entropy. In this program, we use ‘entropy’ because it has a higher accuracy: 0.9615272515301662 than ‘gini’: 0.961284368017099.

b. **Max_depth**: int or None, Optional (default=None) sets the maximum depth of the decision tree in the decision random forest. The greater the depth, the easier the overfitting is. In the project, the max_depth is 15 because the accuracy does not increase anymore when the max_depth becomes higher.

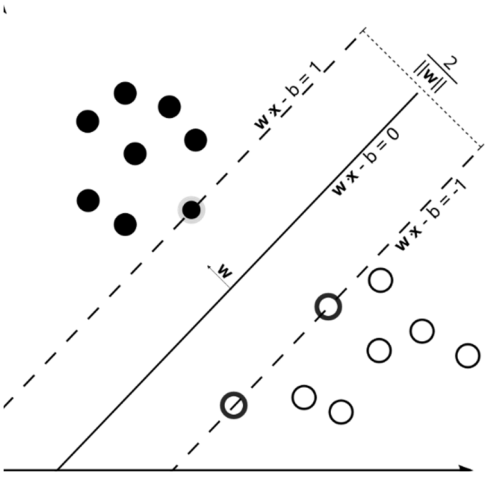
c. **Splitter**: Best or Random. The former is to find the best sharding point among all features, while the latter is to find the best sharding point among some features. As the sample size of the project is large, random is chosen.

d. **min_samples_split**: The minimum number of samples required to split an internal node.

e. **min_samples_leaf**: The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

5. **SVM**: Given a set of training instances, each of which is marked as belonging to one or the other of the two categories, the SVM training algorithm creates a model that assigns new

instances to one of the two categories, making it an probabilistic binary linear classifier.



In this project, the parameters of SVM are:

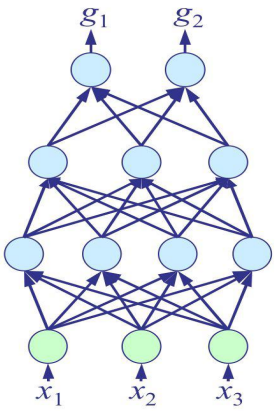
```
SVM :
parameters:  {'C': 1, 'cache_size': 200, 'class_weight': None,
'coef0': 0.0, 'decision_function_shape': 'ovr', 'degree': 3,
'gamma': 'auto_deprecated', 'kernel': 'linear', 'max_iter': -1,
'probability': False, 'random_state': None, 'shrinking': True,
'tol': 0.001, 'verbose': False}
```

a. **C** is the regularization parameter. The strength of the regularization is inversely proportional to C. The project uses its default value 1.

b. **kernel**{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}specifies the kernel type to be used in the algorithm. Here we use 'linear'.

c. **Gamma**: Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

6. **MLP**: Multilayer Perceptron (MLP) is a forward-structured artificial neural network that maps a set of input vectors to a set of output vectors. An MLP can be thought of as a directed graph consisting of multiple node layers, each of which is connected to the next layer.



The parameters:

```
MLP :
parameters:  {'activation': 'identity', 'alpha': 0.0001,
'batch_size': 'auto', 'beta_1': 0.9, 'beta_2': 0.999,
'early_stopping': False, 'epsilon': 1e-08, 'hidden_layer_sizes':
(100,), 'learning_rate': 'constant', 'learning_rate_init': 0.001,
'max_iter': 3000, 'momentum': 0.9, 'n_iter_no_change': 10,
'nesterovs_momentum': True, 'power_t': 0.5, 'random_state': None,
'shuffle': True, 'solver': 'lbfgs', 'tol': 0.0001,
'validation_fraction': 0.1, 'verbose': False, 'warm_start': False}
```

a. **Activation**: Activation function for the hidden layer. Here we use 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x.

b. **Max_iter**: Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. To improve the accuracy, we choose max_iter=3000.

c. **Solver**: The solver for weight optimization. 'lbfgs' is an optimizer in the family of quasi-Newton methods.

d. **early_stopping**: Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation.

e. **learning_rate**: Learning rate schedule for weight updates.

7. **Ensemble**: Different classifiers have different valuable information. To avoid losing potentially valuable information, the project also adopts voting ensemble. The fusion method is continuous-valued output.

   a. **Mean**:

      It calculates the average value of four kinds of classifiers and vote for the higher probability:

```
104 mean_matrix = []
105 mean = (KNN_proba + DT_proba + SVM_proba + MLP_proba) / 4
106 [row,col] = mean.shape
107 for i in range(row):
108     if mean[i][0]>mean[i][1]:
109         mean_matrix.append(1)
110     else:
111         mean_matrix.append(2)
112 mean_accuracy = accuracy_score(y_test, mean_matrix)
```

   b. **Max**:

      It chooses the max value of four classifiers and do the decision based on this max value:

```
temp = []
for j in range(row):
    x = max(KNN_proba[j][0], DT_proba[j][0],SVM_proba[j][0],MLP_proba[j][0])
    y = max(KNN_proba[j][1], DT_proba[j][1],SVM_proba[j][1],MLP_proba[j][1])
    temp.append([x,y])
max_matrix = []
for m in range(row):
    if temp[m][0]>temp[m][1]:
        max_matrix.append(1)
    else:
        max_matrix.append(2)
max_accuracy = accuracy_score(y_test, max_matrix)
```

■ **Requirements:**

The project adopts seven packages: pandas, sklearn.neighbors, sklearn.svm, sklearn.tree, sklearn.neural_network, sklearn.metrics, time:

1. Pandas. Pandas is an open-source, BSD-licensed Python library providing high-perf ormance, easy-to-use data structures and data analysis tools for the Python progra mming language. In the program, pandas is used to read and input the data file 'n ew_data.csv' and 'test_set.csv'.
2. sklearn.neighbors is for installing K-NN.
3. sklearn.svm is for installing SVM.
4. sklearn.tree is for installing decision tree.
5. sklearn.neural_network is for installing MLP.
6. sklearn.metrics for calculating the accuracy of the classifiers.
7. Time is for calculating the training time for each classifier.

■ **Results:**

Classifier	**Accuracy**	Training time
K-NN	0.9669192655202565	0.08002948760986328

Decision Tree	0.9615272515301662	0.030003786087036133
SVM	0.9597784902360827	4.122391700744629
MLP	0.958904109589041	2.1054577827453613
Mean	0.9676479160594579	\
Max	0.9681336830855921	\

```
K-NN :
parameters:  {'algorithm': 'auto', 'leaf_size': 30, 'metric': 'minkowski', 'metric_params':
None, 'n_jobs': None, 'n_neighbors': 5, 'p': 2, 'weights': 'uniform'}
accuracy:  0.9669192655202565
training time:  0.08002948760986328
```

```
Decision Tree :
parameters:  {'class_weight': None, 'criterion': 'gini', 'max_depth': 15, 'max_features':
None, 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None,
'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'presort':
False, 'random_state': None, 'splitter': 'random'}
accuracy:  0.961284368017099
training time:  0.030003786087036133
```

```
SVM :
parameters:  {'C': 1, 'cache_size': 200, 'class_weight': None, 'coef0': 0.0,
'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 'auto_deprecated', 'kernel':
'linear', 'max_iter': -1, 'probability': False, 'random_state': None, 'shrinking': True,
'tol': 0.001, 'verbose': False}
accuracy:  0.9597784902360827
training time:  4.122391700744629
```

```
MLP :
parameters:  {'activation': 'identity', 'alpha': 0.0001, 'batch_size': 'auto', 'beta_1':
0.9, 'beta_2': 0.999, 'early_stopping': False, 'epsilon': 1e-08, 'hidden_layer_sizes':
(100,), 'learning_rate': 'constant', 'learning_rate_init': 0.001, 'max_iter': 3000,
'momentum': 0.9, 'n_iter_no_change': 10, 'nesterovs_momentum': True, 'power_t': 0.5,
'random_state': None, 'shuffle': True, 'solver': 'lbfgs', 'tol': 0.0001,
'validation_fraction': 0.1, 'verbose': False, 'warm_start': False}
accuracy:  0.958904109589041
training time:  2.1054577827453613
```

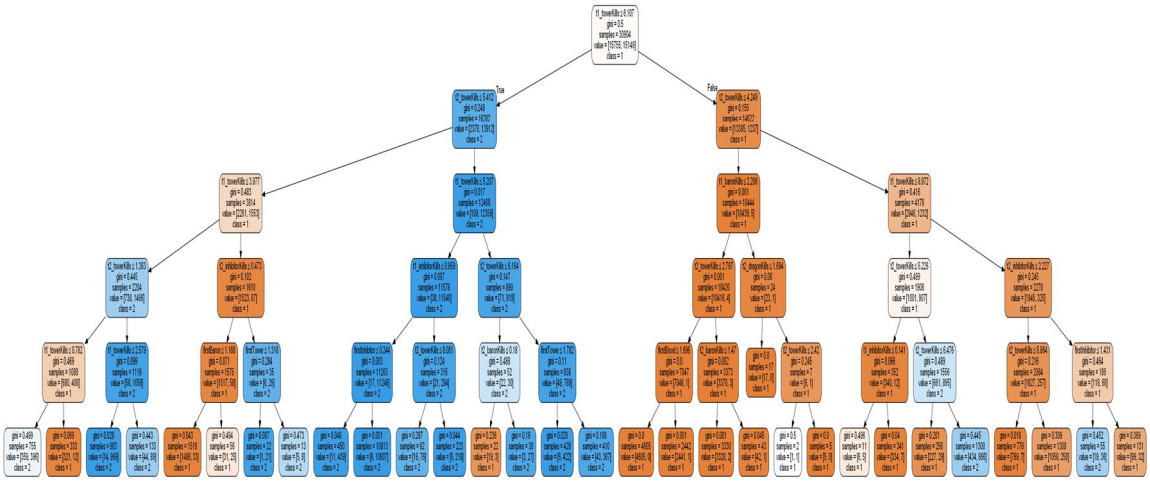
```
Mean :
accuracy:  0.9676479160594579

Max :
accuracy:  0.9681336830855921
```

■ **Comparison and discussion:**

Among all the classifiers in the project, the ensemble with continued-value fusion(max) has the highest accuracy and the Decision Tree has the shortest training time. The ensemble collects all valuable information, so it is more likely to do the classification better. SVM's accuracy is also high but it is too time-consuming because the number of samples is large. MLP's training time is much longer than Decision Tree, too. On the contrary, Decision Tree's accuracy is as high as that of the ensemble but the training time is much shorter. In all, considering both accuracy and efficiency, Decision Tree is the best model for this project and we could use this model to complete prediction in the future.

In my prediction, as the Decision Tree provides explanation on decisions and make decisions based on information gain, we could guess that some certain attributes in the game are prerequisite of the result of the game, which makes the Decision Tree a better classifier in this issue. What's more, we could observe the important fields when we make the tree visible. Here is the visualization of the tree for this project whose depth is five:



According to the tree, we could find that towerKills and inhibitorKills appear most often, so we

could deduce that towerKills and inhibitorKills are most important fields in the game. This could help the creators lay emphasis on these important fields during analysis. The image of the complete tree is in the source file.

For further study and improvement, I will adopt the logistic method. Obviously, this issue is about classification, however, in my opinion, we could generate a series of coordinate systems. For example, each system could represent a kind of field such as first-blood. The new data will be classified by observing which system it fits in best. In this way, we may use the logistic regression to solves this problem. This novel method may do better than the classifiers as it considers more about the details and the law of the statistics of the game. What's more, if more time is allowed, I will do more data preprocessing such as value scaling, feature selection and resampling to improve the data's availability.

■ Conclusion

In conclusion, classification of unlabeled LoL match records with "Win/Loss" could be solved perfectly by machine learning, which helps us know about the reasons behind the statistics in LOL, too. Perhaps during the future research, we could also make use of big data and machine learning to find out better strategies of the games to help our teams win at international events.