Contents

1 Problem Description

1.1 Exercise 1

Implement a function, which takes IDs of two scholars that once worked together as parameters and returns features about the cooperation relationship between them.

Nine possible features are given for reference.

1.2 Exercise 2

Download training and testing files. Then extract features of two scholars in the same line in the training file to construct X matrix and use given labels to construct y vector. With X and y, train a LogisticRegression() model in sklearn and evaluate it with the data in the testing file.

1.3 Exercise 3

Similar to Exercise 2, try to use SVM and Naive Bayes models. Also, try SVM models with different kernels.

Make a comparison between them concerning their training time and prediction accuracy.

1.4 Exercise 4 (Optional)

Implement a simple neural network classifier with Tensorflow and compare it with the previous traditional classifiers.

2 Problem Analysis

2.1 Skills Involved

Firstly, some basic knowledge of SQL is needed to get data from the database. Then, I need to use Python to handle these data. As for the part of machine learning, the usage of numpy, sklearn and tensorflow is required.

Beyond these, an insight into how SVM and CNN work would also help.

2.2 Solution Design

Just like the common procedure of machine learning, extracting features, loading data, training a model and evaluating the model are steps of what I'm going to do.

Whatever machine learning model I choose, extracting features is the common part before training and testing.

Then, for traditional machine learning, I can use Scikit-learn, a useful machine learning package of Python; for deep learning, Tensorflow provides a relatively simple way for me to do the job.

3 Common Part: Extracting Features

To better organize codes, a class named FeatureExtracter() is defined in feature.py, which contains methods like connecting to the database and extracting features. The main parts of it are illustrated as follows:

3.1 Getting Data From Database

First, when an instance of FeatureExtracter() is initialized, the connection to the database is not established.

```python
def __init__(self):
    self.connected = False
```

Then the connect() method is to be called to establish a connection.

```python
def connect(self, user, password, db,
            host="localhost", port=3306, charset="utf8"):
    try:
        self.connection = pymysql.connect(host=host, user=user,
                                          password=password, db=db,
                                          port=port, charset=charset)
        self.cursor = self.connection.cursor()
    except:
        print("Failed to connect to the database!")
    else:
        self.connected = True
```

For convenience, I just get all the papers of thw two scholars in the order of publishing year from the database. The SQL query is like this:

```sql
SELECT paper_author_affiliation.*,papers.paperpublishyear
FROM paper_author_affiliation,papers
WHERE paper_author_affiliation.paperid=papers.paperid
    AND ( paper_author_affiliation.authorid="00000000"
            OR paper_author_affiliation.authorid="11111111")
ORDER BY papers.paperpublishyear ASC,
    paper_author_affiliation.paperid;
```

3.2 Processing Data With Python

In Python, this query's result is a two dimensional list, of which each element is a list containing PaperID, AuthorID, AffiliationID, AuthorSequence and PaperPublishYear. The problem is that the same paper may occur twice in the result if it's done by the two scholars together. So, I need to process the data by myself.

First, the query's result needs to be re-organized. My solution is as follows:

```python
def get_all_papers(self, author1, author2):
    if not self.connected:
        raise RuntimeError("Haven't connected to the database yet!")
    query = "..."   #omitted here
    try:
        self.cursor.execute(query)
    except:
        print("Something unexpected happened when handling the SQL query:")
        print(query)
        self.connection.rollback()
        return ()
    else:
        raw_result = self.cursor.fetchall()
        result = list()
        for row in raw_result:
            if len(result) == 0 or row[0] != result[-1][0]:
                paper_info = [row[0], row[3] if row[1] == author1 else 0,
                              row[3] if row[1] == author2 else 0, row[4]]
                result.append(paper_info)
            else:
                if(row[1] == author1):
                    result[-1][1] = row[3]
                else:
                    result[-1][2] = row[3]
        return result
```

After that, each element of the result is a list of PaperID, IsAuthor1, IsAuthor2, PaperPublishYear.

Then, extract features from the result. Though the given nine features might not be the best, but as far as I'm concerned, they are reasonable enough. Therefore, I just extract these nine features.

When doing this, I make use of some Python features like list comprehension and set to simplify the code:

```python
def extract_feature(self, author1, author2):
    all_papers = self.get_all_papers(author1, author2)
    assert all_papers,"Error! No paper."
    feature = list()
    cooperations = [i for i in range(
        len(all_papers)) if all_papers[i][1] and all_papers[i][2]]
    papers_of_author1 = [i for i in range(
        len(all_papers)) if all_papers[i][1]]
    papers_of_author2 = [i for i in range(
        len(all_papers)) if all_papers[i][2]]
    feature.append(
        len([i for i in range(cooperations[0]) if all_papers[i][1]]))
    feature.append(
        len([i for i in range(cooperations[0]) if all_papers[i][2]]))
    feature.append((feature[0]-feature[1])/len(cooperations))
    feature.append(all_papers[cooperations[0]]
                   [3]-all_papers[papers_of_author1[0]][3])
    feature.append(all_papers[cooperations[0]]
                   [3]-all_papers[papers_of_author2[0]][3])
    feature.append(
        ( feature[3]-feature[4] ) / ( all_papers[cooperations[-1]][3] -
        ↪ all_papers[cooperations[0]][3] + 1))
    feature.append( len ( [i for i in
    ↪ set(papers_of_author1).difference(set(papers_of_author2) )
                        if all_papers[cooperations[0]][3] <= all_papers[i][3] <=
                        ↪ all_papers[cooperations[-1]][3]]))

    feature.append( len ( [i for i in
    ↪ set(papers_of_author2).difference(set(papers_of_author1))
                        if all_papers[cooperations[0]][3] <= all_papers[i][3] <=
                        ↪ all_papers[cooperations[-1]][3]]))
    feature.append((feature[6]-feature[7])/len(cooperations))
    return feature
```

Nearly finished but one thing to note is that the database connection must be closed properly after extracting features. So for this class, a destructor is needed.

```python
def __del__(self):
    if self.connected:
        self.connection.close()
```

4 Solution One: Using Scikit-learn

4.1 Loading Data

For different models in Sklearn, their usage is quite similar. And the part of loading data is the same for following models. As the loading of training data and testing data has no differenece, a function load_data() is implemented.

When load_data() is called, it first checks whether the features have been extracted and saved before . If so, it just loads features from features files by calling load_from_feature_file(). If not, it calls load_from_raw_file(), which extracts features and saves them into certain files for future use. So the code is as follows:

```python
@timer  #attention here
def load_from_raw_file(filepath, feature_extracter, save_into_file=None):
    with open(filepath, mode="r", encoding="utf8") as file:
        X, y = list(), list()
        i = 1
        for line in file:
            print(i, end="\r")
            i += 1
            author1, author2, relation = line.strip().split()
            features = feature_extracter.extract_feature(
                author1, author2)
            X.append(features)
            y.append(int(relation))
        if(save_into_file):
            with open(save_into_file, mode="w", encoding="utf8") as saving_file:
                for feature in zip(y, X):
                    saving_file.write(
                        str(feature[0])+" "+" ".join(map(str, feature[1]))+"\n")
        return X, y

@timer  #attention here
def load_from_feature_file(filepath):
    with open(filepath, mode="r", encoding="utf8") as file:
        X, y = list(), list()
        i = 1
        for line in file:
            print(i, end="\r")
            i += 1
            temp_list = line.strip().split()
            y.append(int(temp_list[0]))
            X.append(list(map(float, temp_list[1:])))
        return X, y

def load_data():
    try:
        extracter = feature.FeatureExtracter()
        extracter.connect("root", "", "academicdb")
    except:
        connected = False
    else:
        connected = True
    if os.path.exists("../Data/train_feature.txt"):
        print("Start to load training data from feature file.")
        X_train, y_train = load_from_feature_file("../Data/train_feature.txt")
    else:
        assert connected
        print("Start to load training data from raw file.")
        X_train, y_train = load_from_raw_file(
            "../Data/train.txt", extracter,
            ↪  save_into_file="../Data/train_feature.txt")
    if os.path.exists("../Data/test_feature.txt"):
        print("Start to load testinging data from feature file.")
        X_test, y_test = load_from_feature_file("../Data/test_feature.txt")
    else:
        assert connected
        print("Start to load testing data from raw file.")
        X_test, y_test = load_from_raw_file(
            "../Data/test.txt", extracter, save_into_file="../Data/test_feature.txt")
    return X_train, y_train, X_test, y_test
```

By the way, as you might have noticed, I also implement a decorator named timer to measure how fast or how slowly a function runs. This decorator can be reused when training models.

```
import time
import functools
def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        tic = time.time()
        result = func(*args, **kw)
        tok = time.time()
        print("Runtime:{:.3f}s".format(tok-tic))
        return result
    return wrapper
```

For example, when loading training data, load_from_feature_file() takes less than 1 second while load_from_row_file() consumes nearly 1 minute. So, it's quite necessary to save features into files.

4.2 Logistic Regresssion

As in the sklearn module premade Logistic Regression Model is provided, things are relatively easy. Just need to import it and use it like these:

```
from sklearn.linear_model import LogisticRegression
@timer
def use_logistic_regression(X_train, y_train, X_test, y_test):
    model = LogisticRegression()
    print("Start to train a logistic regression model.")
    model.fit(X_train, y_train)
    score = model.score(X_test, y_test)
    print("Score of logistic regression:", score)
```

The output information is :

```
Start to train a logistic regression model.
Score of logistic regression: 0.75
Runtime:0.026s
```

4.3 Naive Bayes

Similar to the use of Logistic Regression Model, just import and use Gaussian Naive Bayes Model (or something else like Bernoulli Naive Bayes).

```python
from sklearn.naive_bayes import GaussianNB
@timer
def use_naive_bayes(X_train, y_train, X_test, y_test):
    model = GaussianNB()
    print("Start to train a naive bayes model.")
    model.fit(X_train, y_train)
    score = model.score(X_test, y_test)
    print("Score of naive bayes:", score)
```

The output information is :

```
Start to train a naive bayes model.
Score of naive bayes: 0.720543806647
Runtime:0.016s
```

4.4 SVM

The general usage of SVM in sklearn is also roughly the same as the previous models:

```python
@timer
def use_SVM(X_train, y_train, X_test, y_test, kernel="linear"):
    try:
        model = SVC(kernel=kernel)
        print("Start to train a SVM model(kernel: {0}).".format(kernel))
        model.fit(X_train, y_train)
        score = model.score(X_test, y_test)
        print("Score of SVM(kernel: {0}):".format(kernel), score)
    except:
        print("Error!")
```

4.4.1 Different Kernels

As we know, SVM has different kernels such as linear, rbf and sigmoid. Using different kernel has an impact on the performance of SVM. So I have tried several kernels in this way:

```python
SVM_kernels = ["linear", "rbf", "sigmoid"]
for kernel in SVM_kernels:
    use_SVM(X_train, y_train, X_test, y_test, kernel)
```

The result is as follows:

```
Start to train a SVM model(kernel: linear).
Score of SVM(kernel: linear): 0.730362537764
Runtime:6.807s

Start to train a SVM model(kernel: rbf).
Score of SVM(kernel: rbf): 0.690332326284
Runtime:2.324s

Start to train a SVM model(kernel: sigmoid).
Score of SVM(kernel: sigmoid): 0.615558912387
Runtime:1.207s
```

4.4.2 Optimizing More Parameters

For SVM, there're quite a few parameters to adjust more than just kernel. Among them, c and gamma are two critical parameters. But how to set them to a reasonable value? By cross validation. In sklearn, GridSearchCV() class can help me do th job as it searchs for the best given values of parameters. The code is like these:

```python
def optimize_SVM(X_train, y_train, X_test, y_test):
    C_range = np.logspace(-4, 3, 8)
    gamma_range = np.logspace(-7, 0, 8)
    kernel_range = ["linear", "rbf"]
    param_grid = dict(gamma=gamma_range, C=C_range, kernel=kernel_range)
    grid = GridSearchCV(SVC(),
                        param_grid=param_grid, n_jobs=-1,)
    grid.fit(X_train[:100], y_train[:100])
    print("The best parameters are {0} with a score of {1}".format(grid.best_params_,
    ↪   grid.best_score_))
```

Note that I just take the first 100 groups of features in training data, as too much data would consume an unafford-able amount of time. After about 10 minutes, the result is as follows:

```
The best parameters are {'C': 10.0, 'gamma': 0.0001, 'kernel': 'rbf'} with a score of
↪  0.73
```

Then I use the "best" parameters to train a SVM again, the result is like these, slightly improved.

```
Start to train a SVM model(kernel: rbf).
Score of SVM(kernel: rbf): 0.748489425982
Runtime:0.799s
```

Adjusting the range of parameters and re-optimizing might continue to improve the result, but as it would take too much time, I didn't do that.

4.5 Comparasion

As the result shows, different models, even different parameters make a difference on the performance of training and evaluating. It might be a little arbitray for me to make judgements about them, but some phenomena are evident:

First, Logistic Regression and Naive Bayes are both fast enough when training, while SVMs, especially linear kerneled one, are relatively slow.

Second, as for the prediction accuracy with the default parameters in sklearn, Logistic Regression, Naive Bayes and linear kerneled SVM are at the same level, while other SVMs performs a little disappointingly.

However, with relatively proper parameters, SVM with the rbf kernel wins over SVMs with other kernels, but still at the same level as Logistic Regression and Naive Bayes.

5 Solution Two: Using Tensorflow

5.1 DNN Classifier

When starting to use Tensorflow, I first decide to us a Deep Neural Network Classifier. Considering the size of training data and complexity of problem, two or three layers are enough. If there're too much layers, the performance would not improve but deteriorate.

Before starting to explain my solution, I must make it clear that some of my code consults the official tutorial of Tensorflow to avoid misunderstandings.

5.1.1 Loading Data

As Tensorflow has a premade DNN classifier, tf.estimator.DNNClassifier(), I just make use of it. Then the loading of data also needs to be adjusted as the classifier takes data in dict.

```python
def convert_into_dict(X_raw):
    X = dict()
    for key in range(len(X_raw[0])):
        X[str(key)] = np.array([row[key] for row in X_raw])
    return X

X_train, y_train, X_test, y_test = load_data.load_data()
X_train = convert_into_dict(X_train)
X_test = convert_into_dict(X_test)
```

5.1.2 Training and Testing

This part just follows the usual pattern of Tensorflow and the instructions of Tensorflow's tutorial.

```python
my_feature_columns = []
for key in X_train.keys():
    my_feature_columns.append(tf.feature_column.numeric_column(key=key))
# Build 2 hidden layer DNN with 10, 10 units respectively.
classifier = tf.estimator.DNNClassifier(
    feature_columns=my_feature_columns,
    # Two hidden layers of 10 nodes each.
    hidden_units=[10, 10],
    # The model must choose between 2 classes.
    n_classes=2)
# Train the Model.
classifier.train(
    input_fn=lambda: load_data.train_input_fn(X_train, y_train, 200),
    steps=10000)
# Evaluate the model.
eval_result = classifier.evaluate(
    input_fn=lambda: load_data.eval_input_fn(X_test, y_test, 200))
print('\nTest set accuracy: {accuracy:0.3f}\n'.format(**eval_result))
```

As for train_input_fn() and eval_input_fn() in load_data.py, I also refer to the tutorial:

```python
def train_input_fn(features, labels, batch_size):
    """An input function for training"""
    # Convert the inputs to a Dataset.
    dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))
    # Shuffle, repeat, and batch the examples.
    dataset = dataset.shuffle(1000).repeat().batch(batch_size)
    # Return the dataset.
    return dataset
```

```python
def eval_input_fn(features, labels, batch_size):
    """An input function for evaluation or prediction"""
    features = dict(features)
    if labels is None:
        # No labels, use only features.
        inputs = features
    else:
        inputs = (features, labels)
    # Convert the inputs to a Dataset.
    dataset = tf.data.Dataset.from_tensor_slices(inputs)
    # Batch the examples
    assert batch_size is not None, "batch_size must not be None"
    dataset = dataset.batch(batch_size)
    # Return the dataset.
    return dataset
```

5.1.3 Result

After 10000 steps of training, which takes roughly 30 seconds, the result is as follows:

```
INFO:tensorflow:Saving dict for global step 10000: accuracy = 0.761329,
↪  accuracy_baseline = 0.548338, auc = 0.855578, auc_precision_recall = 0.872124,
↪  average_loss = 0.465701, global_step = 10000, label/mean = 0.548338, loss =
↪  88.0841, precision = 0.809668, prediction/mean = 0.53995, recall = 0.738292
Test set accuracy: 0.761
```

Compared with traditional machine learning methods, DNN takes much more time to train (though in this case it's not very evident due to the small size of data), but has a higher accuracy. (also not very evident for the same reason)

5.2 CNN Classifier

In most cases, Convolutional Neural Network Classifier works better than Deep Neural Network Classifier. Thus I decide to try CNN Classifier for this problem.

5.2.1 Architecture of my CNN

With Tensorflow, a customized CNN Classifier can be constructed for this problem. Also, considering the size of training data and the complexity of the problem, too much layers are useless. So the architecture of my CNN is like these:

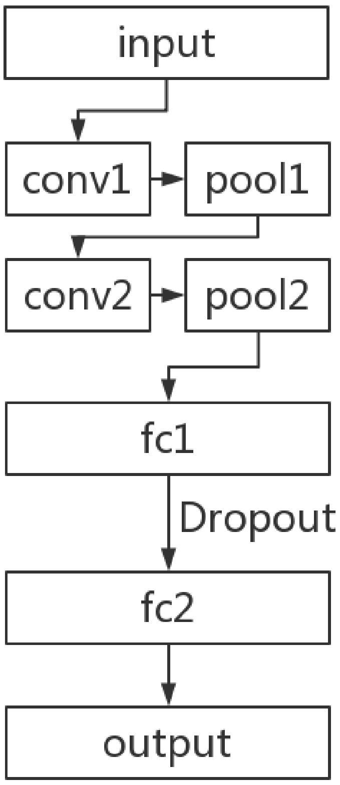

Figure 1: My CNN Architecture

In Python, the code for it is as follows:

```python
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

# convolution
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='VALID')

# pooling
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 1, 1, 1],
     ↪  padding='SAME')

x = tf.placeholder("float", [None, 9])
y_ = tf.placeholder("float", [None, 2])

# first convolutinal layer
w_conv1 = weight_variable([1, 1, 1, 32])
b_conv1 = bias_variable([32])
x_image = tf.reshape(x, [-1, 3, 3, 1])
h_conv1 = tf.nn.relu(conv2d(x_image, w_conv1) + b_conv1)
```

```
h_pool1 = max_pool_2x2(h_conv1)

# second convolutional layer
w_conv2 = weight_variable([1, 1, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, w_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

# densely connected layer
w_fc1 = weight_variable([3*3*64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 3*3*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, w_fc1) + b_fc1)

# dropout
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# readout layer
w_fc2 = weight_variable([1024, 2])
b_fc2 = bias_variable([2])
y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop, w_fc2) + b_fc2)
```

Note that when initializing weight matrix and bias vectors, I define functions to initialize them with small random numbers. The purpose is to avoid problems such as ”Dead Relu”.

5.2.2 Loading Data

When loading data, data need to be preprocessed like normalized. And in accordance with Tensorflow, the label of data should be in the form of one hot vector, like [0,1] and [1,0]. So, I implement a class in load_data.py:

```
class data_set(object):
    def __init__(self):
        X_train, y_train, X_test, y_test = load_data()
        self.y_train = np.array(self.one_hot(y_train))
        self.y_test = np.array(self.one_hot(y_test))
        self.normalizer = preprocessing.Normalizer().fit(np.array(X_train))
        self.X_train = self.normalizer.transform(np.array(X_train))
        self.X_test = self.normalizer.transform(np.array(X_test))

    def one_hot(self, y):
        result = list()
        for yi in y:
            if yi:
                result.append([0, 1])
            else:
                result.append([1, 0])
        return result

    def train_next_batch(self, batch_size):
        mask = np.random.random_integers(
            0, self.X_train.shape[0]-1, batch_size)
        return (self.X_train[mask], self.y_train[mask])
```

5.2.3 Training and Testing

After constructing the CNN, the loss function and optimizer are also needed. For simplicity and efficiency, I choose cross entropy and Adam Optimizer.

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

Then, begin to train and evaluate:

```
sess = tf.InteractiveSession()
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
sess.run(tf.initialize_all_variables())
for i in range(20000):
    batch = data_set.train_next_batch(128)
    if i % 100 == 0:
        train_accuracy = accuracy.eval(
            feed_dict={x: batch[0], y_: batch[1], keep_prob: 1.0})
        print("step {0}, train accuracy {1}".format(i, train_accuracy))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
print("test accuracy {}".format(accuracy.eval( feed_dict = {x: data_set.X_test, y_:
↪  data_set.y_test, keep_prob: 1.0})))
```

5.2.4 Some Hyper-parameters

As is often the case, training a CNN is effected by quite a few hyper-parameters like learning rate, dropout probability, batch size, etc. To achiece better performance, these hyper-parameters need to be optimized carefully.

For example, as for the learning rate, when I switch it to 1e-2, accuracy changes quickly during training but finally stays at a lower number than usual.

5.2.5 Result

Unfortunately, my computer can't afford too much training and thus I can't adjust all the hyper-parameters to the best. So the result of my CNN is not very distinguishing : only 80%, slightly higher than DNN and SVM.

Considering the huge growth of training time, this result is a little disappointing.

6 Conclusion And Prospect

Doing this project, I have gained a better understanding of and a deeper insight into machine learning and deep learning. Considering the scale of the problem, the advantages of deep learning are not evident enough. In fact, in my opinion, for this problem, traditional machine learning models like SVM works even better than DNN and CNN.

But anyway, I have learned tha basic usage of them and I can apply these methods in th future. That's the meaning of this project.