Kaggle/PracticeFusion Competition Documentation

Wendy Kan 10/5/2014

Introduction: My hunt for the prediction variable

I started the task by brainstorming ideas of competitions. From my past experience as a participant and as someone that works in the industry, I think a good competition would have these qualities:

For participants:

- Problem is exciting
- Data is clean and well organized
- Prediction seems like something that's not trivial
- Easy to make a first submission

For companies:

- The prediction has significant business value
- Identify talented candidates and recruit them
- Advertize our data for possible collaborations (API provider/consumers)

Disease and meds - predict a diagnosis.

I started by looking at the data and the past competition. The past challenge was to predict whether a patient has diabetes or not, based on all the features possible (but removing the highly indicative features such as Glucose lab results). This is one big area I can consider - predicting diagnosis.

So I decided to start looking at the most prominent diagnosis in this dataset. Since I work in a drug company (and I know drug related data is very valuable), I'm going to look at medication dataset as well.

I used pandas for my data processing.

In [22]:
```
import pandas as pd
meds = pd.read_csv("/Users/kanc/Documents/classes/Kaggle/practice_fusion/data/trainingS
et/training_SyncMedication.csv")
diag = pd.read_csv("/Users/kanc/Documents/classes/Kaggle/practice_fusion/data/trainingS
et/training_SyncDiagnosis.csv")
diag_meds = meds.merge(diag, on='DiagnosisGuid')

# look at the most frequent diagnosis, list is long, so I'm showing only the first 10
diag_meds['ICD9Code'].value_counts().head(10)
```

Out[22]:
```
401.1     2242
401.9     2117
272.2     1753
250.00    1246
466.0     1219
401        974
530.81     837
724.2      820
477.9      767
244.9      737
dtype: int64
```

As you can see, I have a list of ICD9Code now. What's this 401.1 and 401.9? I did a quick look up at http://en.wikipedia.org/wiki/List_of_ICD-9_codes (http://en.wikipedia.org/wiki/List_of_ICD-9_codes) and found that these are *Hypertension* (high blood pressure). The one following that, 272.2, is *Hyperlipidemia, mixed* (basically high cholesterol). I wanted to group these families of diseases together to get a better sense of them, so I used

```
In [23]: diag_meds['ICD9CodeFamily'] = diag_meds['ICD9Code'].str.split('.').str.get(0)
         diag_meds.ICD9CodeFamily.value_counts().head(10)
```

```
Out[23]: 401    5468
         272    3136
         250    2659
         477    1681
         300    1637
         724    1459
         461    1355
         466    1263
         493    1114
         296    1074
         dtype: int64
```

I'm getting pretty similar results at least for the top 10 disease families. And the number of samples in this set is large enough for a competition, so I have a backup plan for *predicting high cholesterol* or *predicting high blood pressure*. However, I'm not completely satisfied with this competition idea. For a few reasons:

- Predicting a diagnosis is too similar to predicting diabetes. I'll only be asking people to optimize their feature engineering techniques. Not very challenging.
- From a medical point of view, these tests (blood pressure tests, cholesterol level tests) are pretty standard, and not expensive. The cholesterol test is slightly more invasive, but if you are having other tests done, you might as well do that cholesterol test. It doesn't add a lot of value to predict this.

Predicting which drug to use for a specific disease

There are two main tasks a doctor performs: 1. Diagnose 2. Treat. My previous idea mimics the former. I think if a computer can figure out which drug to give to patients given a certain diagnosis, it would mimic (2). And that seems like a cool idea.

So I decided to take a closer look at the hypertension family and figure out how many different types of drugs are given for this condition. If we're lucky, we can build a classifier to predict which drug to use.

```
In [24]: # get the meds that are associated with ICD9 code 401
         hbp_table = diag_meds[diag_meds['ICD9CodeFamily']=='401']
         # group the meds by treatment. See the number of each medication associated with each d
         iagnosis.
         hbp_table.groupby('DiagnosisGuid').size().head()
```

```
Out[24]: DiagnosisGuid
         002550F2-3721-4F77-B2D5-0815CBD05C79    1
         002D1AB1-E643-4BFD-A3B0-3C6BDEFC135A    2
         0038F098-DCE8-4FEF-91A4-6B6A558B5F23    1
         0056BC91-AAFB-43A3-BDDC-8B66C1D49A25    1
         00743005-AEDB-4B37-A184-01E0B7E481E1    2
         dtype: int64
```

Then I realized I have a problem. A lot these diagnosis are linked to multiple drugs. It makes perfect sense because a lot of times all these drugs are given because they want to use this other drug for some other side effects caused by that other drug. There is a lot of inter-dependency here and I don't want to go there.

I figured that the best way is probably to look at only the diagnoses that are only associated with one drug.

```
In [25]: # get how many cases there are where each diagnosis is only associated with one drug
         sum(hbp_table.groupby('DiagnosisGuid').size()==1)
```

```
Out[25]: 1354
```

```
In [26]: # get the most commonly seen drugs for this diagnosis
         hbp_table.NdcCode.value_counts().head(5)
```

```
Out[26]: 115367001    106
         69154041      79
         93111401      78
         69153041      71
         143126730     68
         dtype: int64
```

Ouch! What a bummer. I want my training and test sets to be a little bigger than that. First, I realized there are only 1354 diagnosis associated with only one drug. Then I realized the variety of these drugs is too large, there are very few cases of "most-commonly-prescribed drug" to use from.

It seems like even though it's a great idea, the data I have here is not large enough to have a competitioin for predicting which medication to use for hypertension.

Predicting lab results, smoking status, weight, etc

I started looking also at predicting smoking status (yes/no), but the amount of data in that table is really limited.

I also looked at lab results, hoping I would be able to predict some lab measurements, for example, HDL level. However, I found that in this given data set there are less than 10 samples of the HDL measurements. So lab results is not a great one to run predictions on either.

I looked very briefly at predicting the body weight of each patient. Although this is a fun prediction, it doesn't make much sense because if you have a scale you know your body weight. It's a fun challenge but a meaningless one.

Predicting stats: predicting the number of prescriptions, meds, diagnosis.

Another idea was to calculate some stats and predict those. For example, predict the total number of prescriptions for each patient, predict total number of drugs for each patient, etc. I thought the first idea was pretty cool.

Other than the *coolness*, it also provides a good indicator for medical providers of the person's overall health and risk, for the patient, it gives a good measurement of "they predicted I would have 5 prescriptions but I only have 3". For the competition, it is an interesting measurement that has enough data (at least every patient has some demographic measurement features to start with), and can be extended with more work on feature engineering and multiple levels of modeling (maybe build a map of how many prescriptions are related to each diagnosis).

So the first step is to check on the data. From the previous experience, if I don't have enough sample points, it's not going to work.

In [27]:
```
pat = pd.read_csv("/Users/kanc/Documents/classes/Kaggle/practice_fusion/data/trainingSe
t/training_SyncPatient.csv")
pres = pd.read_csv("/Users/kanc/Documents/classes/Kaggle/practice_fusion/data/trainingS
et/training_SyncPrescription.csv")
# calculate the number of prescriptions each patient
results = pat.merge(pres,on='PatientGuid').groupby('PatientGuid').size()
pat_results = pat.join(results.to_frame(), on='PatientGuid')
pat_results.columns = [u'PatientGuid', u'Gender', u'YearOfBirth', u'State', u'PracticeG
uid', u'prescription_count']
pat_results.head()
```

Out[27]:

	PatientGuid	Gender	YearOfBirth	State	PracticeGuid	prescription_count
0	FB6EFC3D-1A20-4497-9CBD-00027CC5D220	M	1929	SD	7BF4DAD8-5F67-4985-B911-20C9E89A3737	5
1	C6746626-6783-4650-A58F-00065649139A	F	1985	TX	E7101967-2FF1-4B0F-8129-B0B429D1D15C	2
2	E05C6E8F-779F-4594-A388-000C635AE4D3	F	1984	NJ	FC01A799-1CAF-464F-A86F-8A666AB86F32	NaN
3	EAEBD216-F847-4355-87B2-000D942E08F0	M	1959	OH	EEBC95EF-79BE-4542-892E-98D3166BAB20	38
4	C7F10A80-4934-42D2-8540-000FBEBA75C8	F	1990	FL	677BA32E-B4C4-48F2-86E4-08C42B135401	1

This looks pretty good so far. However, I have a decision to make. What about these NaN's? What do they mean? Should I exclude them?

A further look into the data, it becomes obvious that the NaNs are because there's no record associated with this patient in the prescriptions data. Since the minimum number of prescription is 1 and not 0. (Makes sense!)

Note that max is **156**!! That's a lot of prescriptions :(

In [28]:
```
pat_results.prescription_count.describe()
```

Out[28]:
```
count    9116.000000
mean        9.342365
std        13.061517
min         1.000000
25%         2.000000
50%         5.000000
75%        11.000000
max       156.000000
dtype: float64
```

Does that mean I should modify the NaN's to become 0? I don't think that should be the case. There are a lot of reasons for a prescription record to *not* be included in this data. We could simply don't have that record in the system, or maybe it's not logged, maybe the data that Practice Fusion gave me isn't complete, etc. But let's look at one example. I decided to take one of the patients that has a prescription count of NaN, and looked at the diagnosis:

```
In [29]: diag[diag.PatientGuid=='E05C6E8F-779F-4594-A388-000C635AE4D3']
```

Out[29]:

	DiagnosisGuid	PatientGuid	ICD9Code	DiagnosisDescription	StartYear	StopYear	Acute	UserG
52761	3CCED89C-A8F7-458F-815E-89B6B0045AE8	E05C6E8F-779F-4594-A388-000C635AE4D3	401	Essential hypertension ...	NaN	NaN	0	9A10C 1DB4- A649- 13B88
91394	AF0900B3-F57F-441F-B34C-EECD10F93205	E05C6E8F-779F-4594-A388-000C635AE4D3	278.00	Obesity, unspecified	NaN	NaN	0	9A10C 1DB4- A649- 13B88
95965	D96EE03E-D831-40BB-B7F3-FA4A88290BA7	E05C6E8F-779F-4594-A388-000C635AE4D3	V42.0	Kidney replaced by transplant	NaN	NaN	0	9A10C 1DB4- A649- 13B88

Woah! Kidney transplant! Hypertension! It looks like these are pretty critical conditions and there's no way a doctor wouldn't give this patient at least some meds after the transplant. I came to the belief that the data is not complete (missing record, or whatever) and we should simple discard these patient samples if we want to get a good prediction model.

This also made me a bit skeptical of how valid this metric (number of prescription for each patient) is. If not all the prescriptions are logged in the system, can we trust these numbers to be indicator of how risky or how much medical resources a patient needs? After some thoughts, I think the answer is still yes, because we are always going to suffer from *some* data loss, but if we have enough good data, this should still be a good prediction model.

```
In [30]: # drop NAs in the prescription_count column
         pat_results = pat_results.dropna()
         pat_results.head()
```

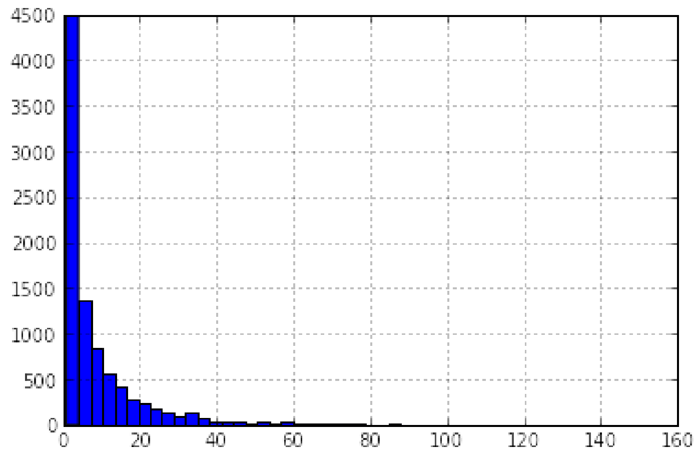
Out[30]:

	PatientGuid	Gender	YearOfBirth	State	PracticeGuid	prescription_count
0	FB6EFC3D-1A20-4497-9CBD-00027CC5D220	M	1929	SD	7BF4DAD8-5F67-4985-B911-20C9E89A3737	5
1	C6746626-6783-4650-A58F-00065649139A	F	1985	TX	E7101967-2FF1-4B0F-8129-B0B429D1D15C	2
3	EAEBD216-F847-4355-87B2-000D942E08F0	M	1959	OH	EEBC95EF-79BE-4542-892E-98D3166BAB20	38
4	C7F10A80-4934-42D2-8540-000FBEBA75C8	F	1990	FL	677BA32E-B4C4-48F2-86E4-08C42B135401	1
5	3BDB6A99-A404-4E9C-BE4B-002054F8B0F4	F	1947	FL	677BA32E-B4C4-48F2-86E4-08C42B135401	2

Also just a quick histogram to visualize the distribution of this prediction count:

```
In [31]: import matplotlib.pyplot as plt
         plt.figure()
         pat_results.prescription_count.hist(bins=50)
```

```
Out[31]: <matplotlib.axes.AxesSubplot at 0x10dd86fd0>
```



Data cleaning

The data exclusion steps I used are the following:

- Removing all the prescription information (training_SyncPrescription.csv) since these are the answers
- Removing all the medication information (training_SyncMedication.csv) since they are too much of an indicator directly related to the number of prescriptions
- Removing all the medication-transcript information (training_SyncTranscriptMedication.csv) since they are easily connected

This is simply removing entire files from the data so doesn't require further data massaging via code.

Splitting of training/test data

Since I have 9116 sets of data, I decided that it is a good idea to just split it in half into training set and test set. I know Kaggle usually uses 30% of test data for public board, and 70% for private board, and that should be included in the detail implementation inside the test set.

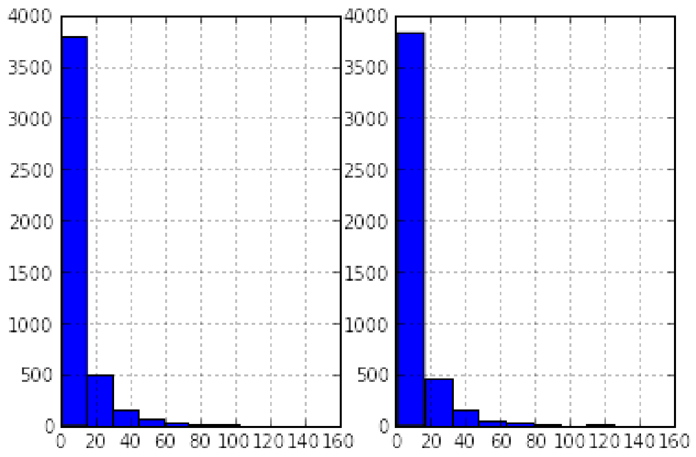
I also had slight concerns about the robustness of my split. In other words, is my split of the train/test set respresentative enough so participants can come up with a reasonaly robust algorithm? In classification data splitting, one can use StratifiedShuffleSplit http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.StratifiedShuffleSplit.html#sklearn.cross_validation.StratifiedShuffleSplit (http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.StratifiedShuffleSplit.html#sklearn.cross_validation.StratifiedShuffleSplit) But in regression it's slightly harder to do. So I made sure that I ran a few iterations of random splits between test/set and that they have similar distribution of the response variable (prediction_count).

Also, I ran these iterations of splits and did a dummy prediction (all 0's) and made sure that the errors (described in the next section) are similar in each run. So we are not doing anything crazy.

```
In [32]: # split the data set into two
         from sklearn.cross_validation import train_test_split
         train, test = train_test_split(pat_results, test_size=0.5, random_state=5)
         traindf = pd.DataFrame(train, columns = pat_results.columns)
         testdf = pd.DataFrame(test, columns = pat_results.columns)


         fig, axs = plt.subplots(1,2)
         traindf.prescription_count.hist(ax=axs[0])
         testdf.prescription_count.hist(ax=axs[1])
```

```
Out[32]: <matplotlib.axes.AxesSubplot at 0x108707290>
```



```
In [33]: # save training set, test set, and solution to csv
         traindf.to_csv('/Users/kanc/Documents/classes/Kaggle/practice_fusion/data/train.csv')
         testdf[testdf.columns - ['prescription_count']].to_csv('/Users/kanc/Documents/classes/K
         aggle/practice_fusion/data/test.csv')
         test_dumb = np.zeros(len(test_true))
         testdf['dumbprediction'] = test_dumb
         testdf[['PatientGuid','dumbprediction']].to_csv('/Users/kanc/Documents/classes/Kaggle/p
         ractice_fusion/data/solution.csv')
```

Evaluation Metric

Since this is a regression problem (predicting a number with range from 1 to around 150), something that's in the family of mean squared error would be the obvious choice. I considered the MSE (Mean Squared Error) and RMSLE (Root Mean Squared Logarithmic Error). Taking a log() on the error definitely changes the scale of the error and can make the numbers become more or less distinguishable between trials.

Since I am running out of time, I chose MSE which is more convenient for me to use from sklearn.

```
In [34]: from sklearn.metrics import mean_squared_error
         test_true = pd.np.array(testdf.prescription_count,dtype=float)
         mean_squared_error(test_true, test_dumb)
```

```
Out[34]: 275.96423870118474
```

Benchmark

At this point I am running short on time. I decided to build a very very simple feature table, taking the first record from the transcripts table with all the vital signs, and use this as a simple feature set to benchmark the competition.

I chose everything out of the box. I used scikit-learn with a random forest regressor. I also tested it with a gradient boosting regressor just to compare.

The results show some improvement from my dummy prediction (which really is dumb, because I used zero which causes a large error). I changed the param for number of estimators to see if I can get better results, and it did.

	Dummy Prediction	**RF n=100**	**RF n=200**	**GBR**
MSE	275.96	195.01	193.82	204.34

I really think it would be a great improvement if I use some other features related to diagnosis, since people with more diagnoses may have more prescriptions. But I have to respect the time constraint and not do that for now.

In [35]:
```
# extract some vitals features from transcripts
trans = pd.read_csv("/Users/kanc/Documents/classes/Kaggle/practice_fusion/data/training
Set/training_SyncTranscript.csv")

# only take the first transcript for each patient
simplefeatures = trans.groupby('PatientGuid').first()
simplefeatures = simplefeatures[[u'Height', u'Weight', u'BMI', u'SystolicBP', u'Diastol
icBP']]
# fill the NAs
simplefeatures = simplefeatures.fillna(simplefeatures.mean())
```

In [36]:
```
# read the training and testing sets
traindf = pd.read_csv('/Users/kanc/Documents/classes/Kaggle/practice_fusion/data/train.
csv',skipinitialspace=1,index_col=0)
testdf = pd.read_csv('/Users/kanc/Documents/classes/Kaggle/practice_fusion/data/test.cs
v',skipinitialspace=1,index_col=0)
# join simple feature set with patient info
traindf = traindf.join(simplefeatures,on='PatientGuid')
testdf = testdf.join(simplefeatures,on='PatientGuid')
# modify gender so RF regressor won't complain
traindf['Gender'] = traindf['Gender'].map( {'F': 0, 'M': 1} ).astype(int)
testdf['Gender'] = testdf['Gender'].map( {'F': 0, 'M': 1} ).astype(int)
```

In [37]:
```
# Train a RFR and predict!
from sklearn.ensemble import RandomForestRegressor
clf = RandomForestRegressor(n_estimators=200)
selected_cols = traindf.columns - ['prescription_count','State','PracticeGuid','Patient
Guid']
clf.fit(traindf[selected_cols], traindf['prescription_count'])
testdf['prediction'] = clf.predict(testdf[selected_cols])
mean_squared_error(test_true, pd.np.array(testdf.prediction,dtype=float))
```

Out[37]:
```
193.82354026436991
```

```
In [38]: # Just to compare, I used a GBR too
         from sklearn.ensemble import GradientBoostingRegressor
         params = {'n_estimators': 100, 'max_depth': 10, 'min_samples_split': 1,'learning_rate':
          0.05, 'loss': 'ls'}
         clf = GradientBoostingRegressor(**params)
         selected_cols = traindf.columns - ['prescription_count','State','PracticeGuid','Patient
         Guid']
         clf.fit(traindf[selected_cols], traindf['prescription_count'])
         testdf['prediction'] = clf.predict(testdf[selected_cols])
         mean_squared_error(test_true, pd.np.array(testdf.prediction,dtype=float))
```

```
Out[38]: 204.33936665155565
```

Conclusion

I've enjoyed participating the competitions, but this is the first time I'm on the other side of the competition! It's a fun, and slightly stressful because of the time constraint, experience.

Lessons learned:

- Most of the questions you asked are interesting, but whether they are possibly achievable is another story
- First look at the data, and then brainstorm. A lot of the ideas I had were not even possible because there are like 5 data points.
- It would be interesting to discuss with the actual data provider (Practice Fusion, in this example), so I can get more insights on what's interesting for them.