Predicting Medical Insurance Claims

New Steps

- One Hot Encoding of categorical predictor variables

-Standardization of continuous output variables

-Neural Nets for regression

-Performance metrics for regression models

-Classic linear regression in sckikit-learn

-The nearest-neighbor predictive model

Previous ideas used again:

-Cross validation to look for evidence of overfitting and estimate expected performance with new data

-Use of permutation methods (ELI5) to understand what the important predictor variables are

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Here is where this data set came from

https://www.kaggle.com/mirichoi0218/insurance?select=insurance.csv

The easiest way to load is typically using pandas, we can convert values into numpy as necessary

```
# Change your infile value

infile="insurance.csv"
med_bills=pd.read_csv(infile)
```

Just a bit of exploratory analysis so we know what we are dealing with here

Content Columns

age: age of primary beneficiary

sex: insurance contractor gender, female, male

bmi: Body mass index, providing an understanding of body, weights that are relatively high or low relative to height, objective index of body weight (kg / m ^ 2) using the ratio of height to weight, ideally 18.5 to 24.9

children: Number of children covered by health insurance / Number of dependents

smoker: Smoking

region: the beneficiary's residential area in the US, northeast, southeast, southwest, northwest.

charges: Individual medical costs billed by health insurance

```
med_bills.head(3)
```

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.90	0	yes	southwest	16884.9240
1	18	male	33.77	1	no	southeast	1725.5523
2	28	male	33.00	3	no	southeast	4449.4620

```
med_bills.shape
```

```
(1338, 7)
```

```
#Notice that the categorical variables don't appear in the describe() function

med_bills.describe()
```

	age	bmi	children	charges
count	1338.000000	1338.000000	1338.000000	1338.000000
mean	39.207025	30.663397	1.094918	13270.422265
std	14.049960	6.098187	1.205493	12110.011237
min	18.000000	15.960000	0.000000	1121.873900
25%	27.000000	26.296250	0.000000	4740.287150
50%	39.000000	30.400000	1.000000	9382.033000
75%	51.000000	34.693750	2.000000	16639.912515
max	64.000000	53.130000	5.000000	63770.428010

pandas data frames can be indexed using the iloc method

the loc member function indexes by the column indices (ie column names) and row names

```
X=med_bills.iloc[:,0:6]
X.head(3)
```

	age	sex	bmi	children	smoker	region
0	19	female	27.90	0	yes	southwest
1	18	male	33.77	1	no	southeast
2	28	male	33.00	3	no	southeast

Set up the target or label for the regresion model

```
y=med_bills.loc[:,'charges']
y.head()
```

```
0    16884.92400
1     1725.55230
2     4449.46200
3    21984.47061
4     3866.85520
Name: charges, dtype: float64
```

Encoding the categorical variables

We have 3 columns which are categorical in nature, sex, smoker, region, and we need to encode these as one-hot variables

The idea is to:

-extract the categorical columns one a time

-Convert them to numpy form (pandas.to_numpy)

-Use the sklearn one-hot-encoder

-Convert the data back into pandas data frames, using the labels from the encoder for the columns

-Concatenate the data frames from the one-hot-encoding back together

I ran into some odd problems with the way the output of the one hot encoder listed the variable names

If you haven't seen Pandas and numpy before, let me know I can point you to some resources on these. Numpy is the matrix structure package, pandas allows for DataFrames like the ones that appear in R, or as tables in SSL

What do we have for groups within these categories

```
med_bills["sex"].unique()
```

```
array(['female', 'male'], dtype=object)
```

```
med_bills["smoker"].unique()
```

```
array(['yes', 'no'], dtype=object)
```

```
med_bills["region"].unique()
```

```
array(['southwest', 'southeast', 'northwest', 'northeast'], dtype=object)
```

This is a one-hot encoding

We create an instance of a OneHotEncoder and then use it to both fit the data med_bills['sex'] and carry out the one-hot encoding

There are a number of pitfalls in this process, which seems so simple.

First, the encoder will try t use numpy sparse matrix encoding by default, this is sometimes handy, but it will cause some unexpected results due to the way the np sparse matrix storage works.

Set sparse in False when you create the instance of the encoder.

Note:This is also called "dummy" coding the variables.

Note that if we have k categories (or levels or groups), we could actually use (k-1) columns, since if the specimen is not in the first (k-1) groups it must be in group k.

I seldom bother to remove the last column, but one could. Removing it may make it more difficult to see variable interactions using permutation or other methods

```
from sklearn.preprocessing import OneHotEncoder

#create an encoder instance

encode_sex=OneHotEncoder(sparse=False)

# extend sex as onehotcolumns.  The conversion to numpy here allows use of the numpy reshape to go to a single complumn variable

mat_sex=encode_sex.fit_transform(med_bills["sex"].to_numpy().reshape(-1,1))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output`
  warnings.warn(
```

```
# the output here is a numpy matrix
# this uses row and column position indexing

mat_sex[0:5,:]
```

```
array([[1., 0.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

We will put this into a pandas dataframe.

The column names are the names used by the encoder, we can see these by looking at the categories_ in the encoder instance

Here comes a weird and annoying point, if we look at this variable, it is a list of one list

```
encode_sex.categories_
```

```
[array(['female', 'male'], dtype=object)]
```

```
type(encode_sex.categories_)
```

```
list
```

```
# notice the oddness right here, the length of encode_sex.categories is 1, but we have two labels.   Look close at the categories_ variable
```

```
len(encode_sex.categories_)
```

```
1
```

```
encode_sex.categories_[0][0]
```

```
'female'
```

Why does this matter? I want to use these titles as the column labels,but if I use a list of list for the columns, we creata a Multiindex, a new feature in the pandas dataframes

a Multiindex is a hierarchical column name, which probably has uses

Watch what happens here if we set the column names to encode_sex.categories_

```
df_sex=pd.DataFrame(mat_sex,columns=encode_sex.categories_)
```

```
df_sex.columns
```

```
MultiIndex([('female',),
            (  'male',)],
           )
```

Instead of the column numes being just a list, they are a thing called a Multiindex by setting the column names to a list of lists

We don't want this, so here's the syntax to get only the first time in the list

```
df_sex=pd.DataFrame(mat_sex,columns=encode_sex.categories_[0])
```

```
df_sex.columns
```

```
Index(['female', 'male'], dtype='object')
```

I spent a long time figuring out what was happening here, really annoying

```
# check to see df_sex has what we want
df_sex.head()
```

	female	male
0	1.0	0.0
1	0.0	1.0
2	0.0	1.0
3	0.0	1.0
4	0.0	1.0

```
encode_smoker=OneHotEncoder()
```

```
mat_smoker=encode_smoker.fit_transform(med_bills["smoker"].to_numpy().reshape(-1,1))
```

```
df_smoker=pd.DataFrame(mat_smoker.toarray(),columns=encode_smoker.categories_)
```

```
encode_smoker.categories_
```

```
[array(['no', 'yes'], dtype=object)]
```

```
# manually set column names,  "no" and "yes" don't mean much,   we want more informative column names
df_smoker.columns=["Non_smoker","Smoker"]


# just checking!
df_smoker.head()
```

	Non_smoker	Smoker
0	0.0	1.0
1	1.0	0.0
2	1.0	0.0
3	1.0	0.0
4	1.0	0.0

```
encode_region=OneHotEncoder()

mat_region=encode_region.fit_transform(med_bills["region"].to_numpy().reshape(-1,1))

df_region=pd.DataFrame(mat_region.toarray(),columns=encode_region.categories_[0])

df_region.head()
```

	northeast	northwest	southeast	southwest
0	0.0	0.0	0.0	1.0
1	0.0	0.0	1.0	0.0
2	0.0	0.0	1.0	0.0
3	0.0	1.0	0.0	0.0
4	0.0	1.0	0.0	0.0

```
df_region.columns
```

```
Index(['northeast', 'northwest', 'southeast', 'southwest'], dtype='object')
```

```
# concatenate these one-hot-encoded versions of the categorical variables, and check them

df_cats=pd.concat([df_sex,df_smoker,df_region], axis=1)
df_cats.head()
```

	female	male	Non_smoker	Smoker	northeast	northwest	southeast	southwest
0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0
1	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0
2	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0
3	0.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0
4	0.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0

```
df_cats.columns
```

```
Index(['female', 'male', 'Non_smoker', 'Smoker', 'northeast', 'northwest',
       'southeast', 'southwest'],
      dtype='object')
```

It looks like we now have to be a bit careful about this issue of Multiindexes, this is a new "feature".

Standardization

We will standardize the continuous variables within a pipeline. A pipeline carries out the action requested on each column of the data input

A pipeline can contain many cleaning and formatting operations, we will both standardize and impute here. We don't really need to imputer in this case, the impute was added as an example

In the standard scaler, we will subtract the mean value from each column, and then divide the column by its own standard deviation.

This forces all the variables to be on the same "scale" and keeps them all close to zero where rounding errors are lowest.

```
med_bills.head()
```

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('std_scaler', StandardScaler()),
    ])

# pipeline only the continuous data

med_bill_continuous= num_pipeline.fit_transform(med_bills[['age','bmi','children']])

# put the resulting numpy matrix back into a dataframe

med_bill_continuous=pd.DataFrame(med_bill_continuous,columns=['age','bmi','children'])


# check to be sure this worked right

med_bill_continuous.head()
```

	age	bmi	children
0	-1.438764	-0.453320	-0.908614
1	-1.509965	0.509621	-0.078767
2	-0.797954	0.383307	1.580926
3	-0.441948	-1.305531	-0.908614
4	-0.513149	-0.292556	-0.908614

```
#concatenate with the categorical variables

med_bill_final=pd.concat([med_bill_continuous,df_cats],axis=1)

med_bill_final.head()
```

	age	bmi	children	female	male	Non_smoker	Smoker	northeast	northwest
0	-1.438764	-0.453320	-0.908614	1.0	0.0	0.0	1.0	0.0	0.0
1	-1.509965	0.509621	-0.078767	0.0	1.0	1.0	0.0	0.0	0.0
2	-0.797954	0.383307	1.580926	0.0	1.0	1.0	0.0	0.0	0.0
3	-0.441948	-1.305531	-0.908614	0.0	1.0	1.0	0.0	0.0	1.0
4	-0.513149	-0.292556	-0.908614	0.0	1.0	1.0	0.0	0.0	1.0

test and train split

We have the data set "munged" (formatted really) the way we want it, scaled and one-hot-encoded, we can now form the test and train set.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(med_bill_final, y,train_size=0.8,random_state=1)
```

```
y_train
```

```
216     10355.64100
731     10065.41300
866      1141.44510
202     13012.20865
820      7445.91800
           ...
715     12146.97100
905      4564.19145
1096    44641.19740
235     19444.26580
1061    11554.22360
Name: charges, Length: 1070, dtype: float64
```

```
X_train
```

	age	bmi	children	female	male	Non_smoker	Smoker	northeast	northw
216	0.982076	-0.666578	-0.908614	1.0	0.0	1.0	0.0	0.0	
731	0.982076	-1.519609	-0.078767	0.0	1.0	1.0	0.0	0.0	
866	-1.509965	1.087058	-0.908614	0.0	1.0	1.0	0.0	0.0	
202	1.480485	-1.087352	-0.908614	1.0	0.0	1.0	0.0	0.0	
820	0.412467	0.498138	-0.078767	0.0	1.0	1.0	0.0	0.0	
...	...	...	...	...	...	...	...	...	
715	1.480485	-0.289276	-0.908614	0.0	1.0	1.0	0.0	0.0	
905	-0.940356	-0.214635	0.751079	1.0	0.0	1.0	0.0	1.0	
1096	0.839674	0.704834	0.751079	1.0	0.0	0.0	1.0	1.0	
235	0.056461	-1.385093	0.751079	1.0	0.0	0.0	1.0	0.0	
1061	1.266881	-0.446758	-0.078767	0.0	1.0	1.0	0.0	0.0	

1070 rows × 11 columns

Save the data

At this point on a serious project, I would save the test and train data separately, and close the file. I'd start the modeling in a new notebook or file, just to avoid any chance of contaminating the project wit the test data

I'd also do an exploratory data analysis, to see what is going on in the data set, and only then go on to building predictive models.

To save time, we are just going to dig into the models

-Neural Net regressor

-a linear regression model

-a nearest-neighbor model

Build the model

In scikit learn, all models work the same way, in terms of setup

-create a model instance

-fit or train it

-make predictions

The MLPRegressor uses a mean squared error (MSE) criterion for the Loss

We can make use of a variety of performance metrics to understand how the model is doing. These are resubstitution estimates made on the training data, which tend to be over-optimistic due to the prevalence of overfitting

```
from sklearn.neural_network import MLPRegressor

regr = MLPRegressor(hidden_layer_sizes=(6,3,),random_state=1, max_iter=50000, verbose=False)
regr.fit(X_train, y_train)
y_pred=regr.predict(X_train)
```

```
from sklearn.metrics import explained_variance_score

explained_variance_score(y_train,y_pred)
```

```
0.84175188769289
```

```
from sklearn.metrics import mean_squared_error

mean_squared_error(y_train, y_pred)
```

```
23081098.960187037
```

```
 from sklearn.metrics import mean_absolute_percentage_error

mean_absolute_percentage_error(y_train, y_pred)
```

```
0.2882433763829543
```

```
from sklearn.metrics import r2_score
r2_score(y_train, y_pred)
```

```
0.8417465281140449
```

Try a linear regresion model, classic least square regression

Again it fits and trains the same way

```
from sklearn.linear_model import LinearRegression

reg = LinearRegression().fit(X_train,y_train)
y_pred_lin=reg.predict(X_train)
```

```
explained_variance_score(y_train,y_pred_lin)
```

```
0.7477680686451552
```

Try nearest neighbor regression

This almost isn't a model, it simply finds the k specimens closest to a specimens and uses the average value of the k nearest neighbors as the regression value- Note that you can build classifers this way as well.

The k value used here was 8, this is a hyperparamenter

```
from sklearn.neighbors import KNeighborsRegressor

neigh = KNeighborsRegressor(n_neighbors=8)
neigh.fit(X_train, y_train)

y_pred_nn=neigh.predict(X_train)

explained_variance_score(y_train,y_pred_nn)
```

```
0.8055459624821658
```

```
r2_score(y_train,y_pred_nn)
```

```
0.8031959620565939
```

Cross validation

Use cross validation to determine what the expected r2 value would be on new data

An example is shown for the nearest neighbor model

I am going to change cv = 4 becuase cv= 10 for the nural net is taking 30 mins.

```
from sklearn.model_selection import cross_val_score

neigh2 = KNeighborsRegressor(n_neighbors=8)
scores = cross_val_score(neigh2, X_train, y_train, cv=10,scoring='r2')
```

```
np.mean(scores)
```

```
0.7229553759141347
```

```
np.std(scores)
```

```
0.050203059757298704
```

```
scoresTest = cross_val_score(neigh2, X_test, y_test, cv=10,scoring='r2')
```

```
np.mean(scoresTest)
```

```
0.6480823107908119
```

```
np.std(scoresTest)
```

```
0.15694708037126823
```

Question: Use cross validation to examine the performance of the 2 other models, linear regression and the Neural net.

Which models seem to be overfit? Can you adjust them slightly to improve performance (don't spend all night on this!)

Regression

```
reg2 = LinearRegression().fit(X_train,y_train)
scores2 = cross_val_score(reg2, X_train, y_train, cv=4,scoring='r2')
```

```
np.mean(scores2)
```

```
0.740877587661711
```

```
np.std(scores2)
```

```
0.019272437166336205
```

```
scores2Test = cross_val_score(reg2, X_test, y_test, cv=4,scoring='r2')
```

```
np.mean(scores2Test)
```

```
0.7262739138547012
```

```
np.std(scores2Test)
```

```
0.08521015226387571
```

Nural Net

```
regr2 = MLPRegressor(hidden_layer_sizes=(6,3,),random_state=1, max_iter=50000, verbose=False)
Score3Input = regr2.fit(X_train, y_train)
scores3 = cross_val_score(Score3Input, X_train, y_train, cv=4,scoring='r2')
```

```
np.mean(scores3)
```

```
0.7445978859573651
```

```
np.std(scores3)
```

```
0.022954821628364985
```

```
scores3Test = cross_val_score(Score3Input, X_test, y_test, cv=4,scoring='r2')
```

```
np.mean(scores3Test)
```

```
0.6868180509440396
```

```
np.std(scores3Test)
```

```
0.07882859324052291
```

KNeighborsRegressor is the most overfitted as the average R^2 from the testing set is higher than the training set by .11, which is by far the greatest diffrence. This is when the cv = 4 the diffrence between the testing and training set decreases as I increase the cv. However the simpletst model the linear regression model in this case in the least overfitted at cv = 4 so we shouldn't bother with the other models.

Question: Permutation estimates of variable importance

For each of the three models, use the permutation tools to determine the importance of the different variables.

Can you find strong evidence of interaction of variables?

```
#install eli5 when running in google colab
```

```
!pip install eli5
```

```
ing eli5
oading eli5-0.13.0.tar.gz (216 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 216.2/216.2 kB 4.4 MB/s eta 0:00:00
ring metadata (setup.py) ... done
ment already satisfied: attrs>17.1.0 in /usr/local/lib/python3.10/dist-packages (from eli5) (23.2.0)
ment already satisfied: jinja2>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from eli5) (3.1.3)
ment already satisfied: numpy>=1.9.0 in /usr/local/lib/python3.10/dist-packages (from eli5) (1.23.5)
ment already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from eli5) (1.11.4)
ment already satisfied: six in /usr/local/lib/python3.10/dist-packages (from eli5) (1.16.0)
ment already satisfied: scikit-learn>=0.20 in /usr/local/lib/python3.10/dist-packages (from eli5) (1.2.2)
ment already satisfied: graphviz in /usr/local/lib/python3.10/dist-packages (from eli5) (0.20.1)
ment already satisfied: tabulate>=0.7.7 in /usr/local/lib/python3.10/dist-packages (from eli5) (0.9.0)
ment already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2>=3.0.0->eli5) (2.1.5)
ment already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.20->eli5) (1.3.2)
ment already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.20->eli5) (3.2.0)
g wheels for collected packages: eli5
ing wheel for eli5 (setup.py) ... done
ed wheel for eli5: filename=eli5-0.13.0-py2.py3-none-any.whl size=107717 sha256=0a9733004d11391e330eecea9f30202f1d57dc5b13192cd06f783f62
d in directory: /root/.cache/pip/wheels/b8/58/ef/2cf4c306898c2338d51540e0922c8e0d6028e07007085c0004
fully built eli5
ing collected packages: eli5
fully installed eli5-0.13.0
```

```
med_bill_final.columns
```

```
      Index(['age', 'bmi', 'children', 'female', 'male', 'Non_smoker', 'Smoker',
             'northeast', 'northwest', 'southeast', 'southwest'],
```

```
import eli5
from eli5.sklearn import PermutationImportance
```

```
perm = PermutationImportance(neigh, random_state=1).fit(X_train, y_train)
```

```
eli5.show_weights(perm, feature_names = list(med_bill_final.columns))
```

Weight	Feature
0.3489 ± 0.0178	Smoker
0.3468 ± 0.0373	Non_smoker
0.1629 ± 0.0268	age
0.1486 ± 0.0427	bmi
0.0321 ± 0.0204	children
-0.0069 ± 0.0108	southeast
-0.0115 ± 0.0064	northwest
-0.0130 ± 0.0080	northeast
-0.0170 ± 0.0051	southwest
-0.0173 ± 0.0059	female
-0.0187 ± 0.0076	male

```
perm2 = PermutationImportance(reg2, random_state=1).fit(X_train, y_train)
```

```
eli5.show_weights(perm2, feature_names = list(med_bill_final.columns))
```

Weight	Feature
0.3185 ± 0.0133	Smoker
0.3155 ± 0.0351	Non_smoker
0.1765 ± 0.0334	age
0.0532 ± 0.0106	bmi
0.0042 ± 0.0016	children
0.0006 ± 0.0009	northeast
0.0006 ± 0.0013	southeast
0.0002 ± 0.0007	southwest
0.0002 ± 0.0004	northwest
0.0001 ± 0.0004	male
0.0001 ± 0.0001	female

```
perm3 = PermutationImportance(regr2, random_state=1).fit(X_train, y_train)
```

```
eli5.show_weights(perm3, feature_names = list(med_bill_final.columns))
```

Weight	Feature
0.6722 ± 0.0294	Smoker
0.3036 ± 0.0348	Non_smoker
0.2260 ± 0.0420	bmi
0.1956 ± 0.0254	age
0.0414 ± 0.0047	northeast
0.0369 ± 0.0027	female
0.0361 ± 0.0086	northwest
0.0255 ± 0.0071	southwest
0.0239 ± 0.0068	southeast
0.0225 ± 0.0073	male
0.0079 ± 0.0017	children

Smoker and non smoker are the 2 most imopirtant in all of them, age is the 3rd most important in nearest neighbour regression and age is the 3rd most important in the MLP regressor.

```
Start coding or generate with AI.
```