Raw Data - Per Patient Grouping

The native structure of the ADNI data is a longitudinal form with one entry per visit, with each visit having an associated visit vode. Ultimately we want a data set with one entry per patient. An important decision will be determining how to split up the data. The purpose of this notebook is to explore methods of building a per patient data set.

Import libraries

In [1]:

```
%matplotlib inline
import os
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

# import custom dependencies
from ADNI_utilities import define_terms, describe_meta_data, paths_with_ext, append_meta_cc
```

Import data

The data used in this notebook will be a combination of raw ADNI data files previously cleaned and curated for this project. The data included falls into three categories:

Biomarkers (discussed in detail in `biomarker_EDA.ipynb`)

- ApoE gene data `apoe_clean.csv`
- Amyloid-beta, Tau, and pTau levels measured from cerebral spinal fluid `csf_clean.csv`
- Laboratory chemical screenings of patient blood and urine `lab_clean.csv`

Medical History (discussed in detail in `medical_history_EDA.ipynb`)

- neurological exams `neuroexams_clean.csv`
- general medical history `medhist_clean.csv`
- baseline health assessment `bls_symptoms.csv`
- vital signs measurments `vitals.csv`

Subject Characteristics (discussed in detail in `sub_characteristics.ipynb`)

- family history of dementia `famhist_clean.csv`
- demographics `demographics_clean.csv`

Neuropsychological Assessments (discussed in detail in `neuropsych_scores.ipynb`)

- geriatric depression scale `depression_clean.csv`
- mini-mental state exam `mmse_clean.csv`

- modified hachinski ischemia exam `mhach_clean.csv`

Diagnostic Data (discussed in detail in `Response_var_EDA.ipynb`)

- patient diagnoses including, but not limited to, Alzheimer's diagnosis `diagnosis_clean.csv`

In [2]:

```
# import adni dictionary
apo_dict = pd.read_csv("../data/Biomarker Data/APOERES_DICT.csv")
```

Data Grouping

Breakdown by VISCODE

One possibility for aggregating the data is to take the first baseline (code: `bl`) visit for each patient from each data set. This is an attractive possibility because the merged data table provided by ADNI (adnimerge) has a unique entry per patient per visit code. We can look at the breakdown of the number of observations per patient separated by `VISCODE` to see if this approach is viable.

To start let's define a function to return the number of records for each patient.

In [3]:

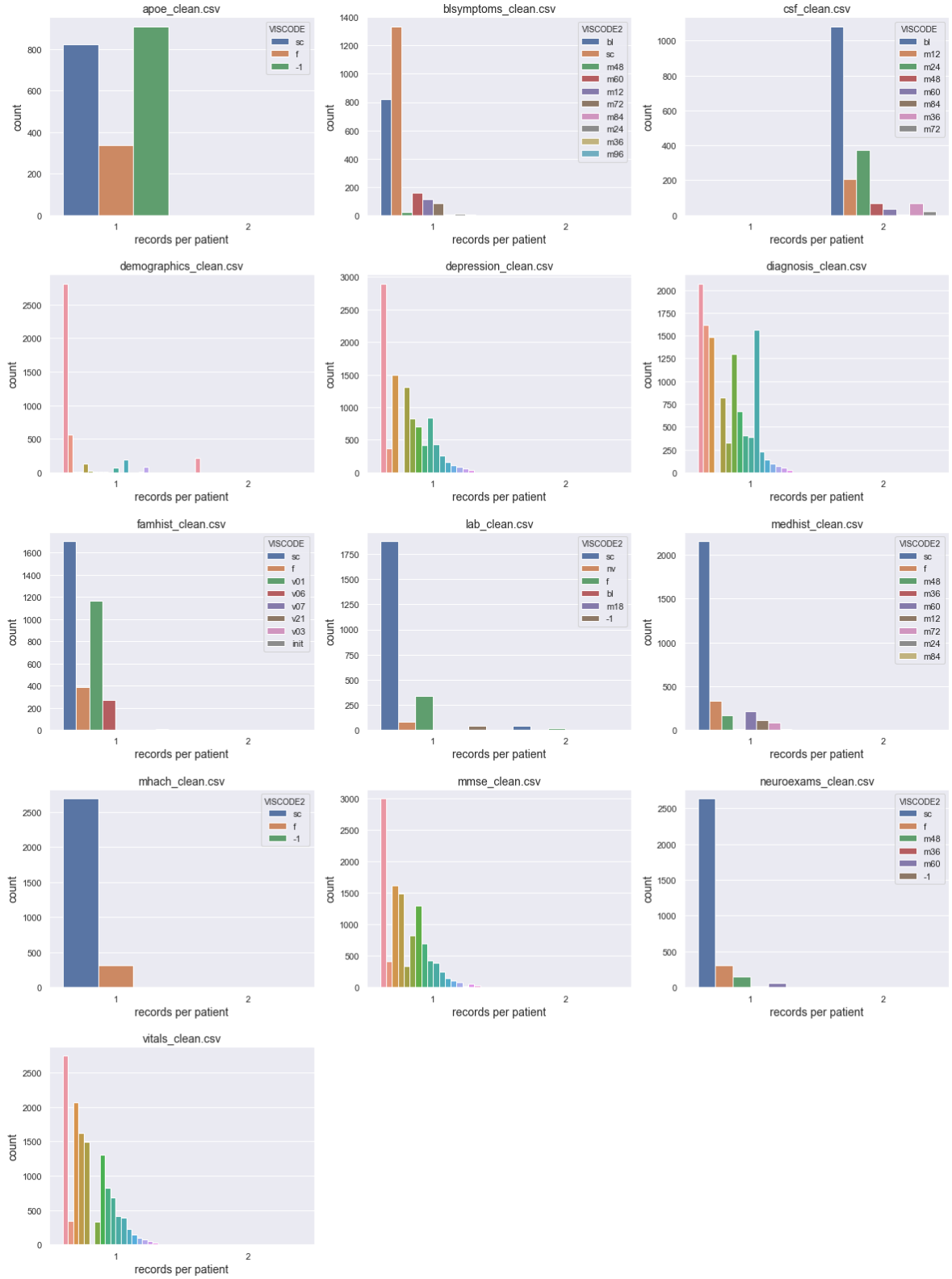
```
# function that returns the number of records per patient from a dataframe
def patient_num_records(df):

    # get indices grouped by patient ID (RID)
    n_measurements = df.groupby("RID").apply(lambda x: x.shape[0])

    return(n_measurements)
```

Now we can iterate over each raw data file and look at the distribution of number of records per patient in each `VISCODE`. We'll also display the visit codes in the legend to get a sense of which visit codes are available in each data set. As we will see, some of the files contain too many unique visit codes to display. Because we are really only interested in whether or not the data contains a unique record for each visit code, we will visualize the data as one record per patient and more than one record.

In [54]:

```
# define group and bins
grp = "VISCODE2"
bins = np.arange(1,4,1)
nbins = bins.shape[0]-1

# get data paths
csv_paths = paths_with_ext(directory="../data/Cleaned/")

# configure subplots
nrows = 5
ncols = np.ceil(len(csv_paths)/nrows)

# iterate over dataframes
plt.figure(figsize=(20,28))

for i, path in enumerate(csv_paths):

    # read in current dataframe
    df = pd.read_csv(path, low_memory=False)

    # create subplot for histogram of each df
    if "VISCODE2" in df.columns:
        grp="VISCODE2"
    elif "VISCODE" in df.columns:
        grp = "VISCODE"
    else:
        grp = False

    # if the file contains visit code meta data
    if grp:

        # group data by visit code and count number of records per patient in each
        by_viscode = df.groupby(grp)
        records_per_viscode = by_viscode.apply(patient_num_records)

        # cap number of records per patient at 2 (for visualization)
        #records_per_viscode[records_per_viscode>2] = 2

        # get viscodes from df
        viscodes = df[grp].dropna().unique()
        nv = viscodes.shape[0]
        vc = np.tile(viscodes,nbins)
        tmp_counts = np.zeros(viscodes.shape[0]*nbins)
        tmp_n = np.tile(np.arange(1,nbins+1,1).reshape(-1,1),nv).reshape(-1,nv).flatten()
        tmp_df = pd.DataFrame(data=np.vstack((vc,tmp_n,tmp_counts)).T, columns=[grp,"record

        # plot the data
        plt.subplot(nrows,ncols,i+1)
        for j,code in enumerate(viscodes):

            # count num in each category
            tmp_df.loc[tmp_df[grp]==code,'count'] = np.histogram(records_per_viscode[code].

        # plot histogram
        sns.barplot(x="records", y="count", hue=grp, data=tmp_df)
        plt.xlabel("records per patient", FontSize=14)
        plt.ylabel("count", FontSize=14)
        plt.title("{}".format(os.path.basename(path)), FontSize=14)
```

```
        # display legend there are few than 13 unique codes
        if viscodes.shape[0] > 10:
            ax = plt.gca()
            ax.get_legend().set(Visible=False)


plt.subplots_adjust(hspace=0.3)
```



We can see from the plots above that not all data sets will have a `bl` visit or will only be sparsely represented. Additionally, some data sets will have multiple `bl` records for each patient. In the interest of completeness, we should consider another strategy to aggregate the data.

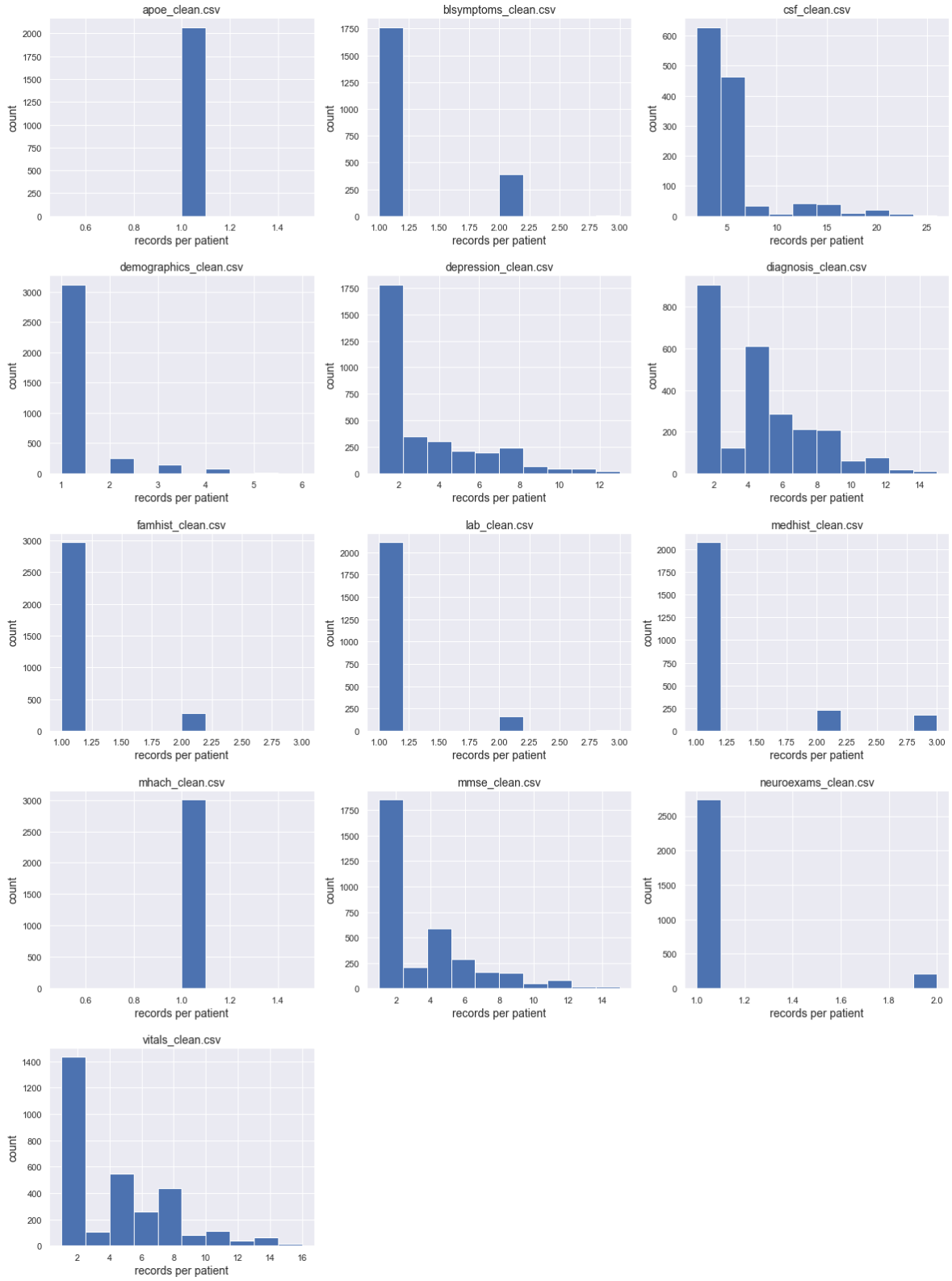
Records per patient

With the ultimate goal being to create a data set with the format of one entry per patient, we will need to find approaches of collapsing the various measurements from each visit for each patient into a single entry. With this goal in mind, it should be helpful to look at the distribution of number of entries per patient for each data set.

Plot the distribution of entry number per patient for each dataframe

In [61]:

```
# iterate over dataframes
plt.figure(figsize=(20,28))
for i, path in enumerate(csv_paths):

    # read in current dataframe
    df = pd.read_csv(path, low_memory=False)

    # get number of records per patient
    n = patient_num_records(df)

    # plot histogram of results
    plt.subplot(nrows,ncols,i+1)
    plt.hist(n)
    plt.xlabel("records per patient", FontSize=14)
    plt.ylabel("count", FontSize=14)
    plt.title(os.path.basename(path), FontSize=14)

plt.subplots_adjust(hspace=0.3)
```



As we can see the distribution of number of records per patient varies quite a lot. There are many different approaches we could take to compile a single entry. Simple strategies include:

1. Take the first entry per patient
2. Build the most complete entry for each patient (taking the first non-missing entry for each patient and feature)
3. Take per patient average for numeric data or per patient mode for categorical data

For data sets with only a single entry per patient, we only have one option: take the first and only record for each patient. Two of our data sets (ApoE and MHach) fall into this category, but most of our data sets could potentially stand to benefit from a broader approach. We can also see that even in data sets where multiple

entries per patient occur, only a small subset of the patients actually have multiple entries. For those reasons, I prefer approaches that take a single measurement per patient per feature, to make the measurements more comparable across patients. This leaves the first two strategies above.

To assess whether the strategy of building a most complete entry is viable, we should see if patients with missing values in the first entry have the values filled in other entries. One way to compare this two approaches is to define functions to build entries both ways and profile missingness in the resulting data set.

In [58]:

```
# define a function to extract the index-th entry for each patient from a datafame(df)
def patient_ith_entry(df, index=0):

    # group by patient ID
    by_RID = df.groupby("RID")

    # extract the df index for the index-th row for each patient
    entry_idx = by_RID.apply(lambda x: x.iloc[index]["Unnamed: 0"]).values
    entry_dat = df.iloc[entry_idx]
    entry_dat.set_index("RID", inplace=True)
    return(entry_dat, entry_idx)
```

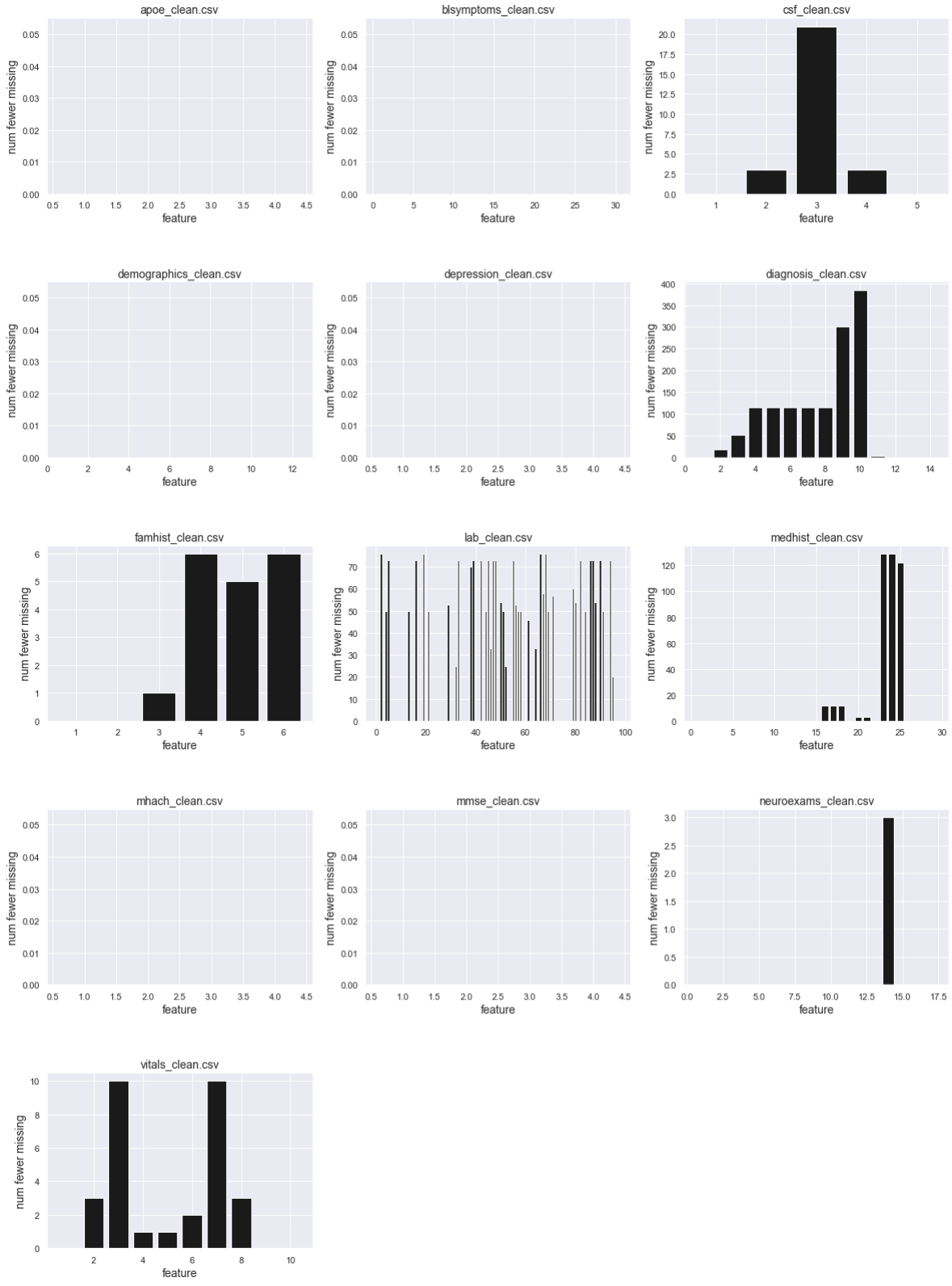
In [59]:

```
# define a function to extract the first non-nan entry for each patient and feature
def patient_first_nonmissing_entry(df, missing_val=-1, index=0):

    # group by RID
    by_RID = df.groupby("RID")

    # define function to extract first non-missing val from series
    if np.isnan(missing_val):
        get_first_nonmissing = lambda y: y.loc[~y.isna()].iloc[0] if any(~y.isna()) else mi
    else:
        get_first_nonmissing = lambda y: y.loc[y!=missing_val].iloc[0] if any(y!=missing_va

    # for each RID grouping, apply function to each column
    by_nonmissing = by_RID.apply(lambda x: x.apply(get_first_nonmissing))
    return(by_nonmissing.drop(columns="RID", axis=1))
```

Now populate two per patient dataframes using the above methods and compare the amount of missingness between the two by plotting the number of missing values in each column. Display the difference in number missing between the two strategies.

In [62]:

```
# initialize placeholders for new dataframes
first_idx_dfs = []
nonmissing_dfs = []
missing_value = -1

# iterate over dataframes
plt.figure(figsize=(20,28))
for i, path in enumerate(csv_paths):

    # read in current dataframe
    df = pd.read_csv(path, low_memory=False)

    # extract first entry
    RID_df, _ = patient_ith_entry(df)
    first_idx_dfs.append(RID_df)
    nmiss_first = first_idx_dfs[i].apply(lambda x: np.sum(x==missing_value))

    # extract first non-missing entries
    nonmissing_dfs.append(patient_first_nonmissing_entry(df))
    nmiss_non = nonmissing_dfs[i].apply(lambda x: np.sum(x==missing_value))

    # plot missingness difference between the two grouping methods
    plt.subplot(nrows,ncols,i+1)
    plt.bar(np.arange(1,nmiss_first.shape[0]+1,1), nmiss_first.values - nmiss_non.values, c
    plt.ylim(0,plt.ylim()[1])
    plt.xlabel("feature", FontSize=14)
    plt.ylabel("num fewer missing", FontSize=14)
    plt.title(os.path.basename(path), FontSize=14);

plt.subplots_adjust(hspace=0.5)
```



In at least one data set (lab data), taking the first non-missing value recovers some additional data for a subset of the features, although it's not a huge effect compared to the number of patients in the data set (≈3000). The penalty we pay for this additional data is that the data is more likely to be from diverse time points throughout the study.

Final Diagnosis

For the purposes of our analysis, we will likely want to track the change in a patient's diagnosis from the baseline assessment to the final visit. We can use the method described above to take the patient entry at a particular index. This time we'll use `index = -1` to take the last entry for each patient. We'll also return the

row index within the dataframe for both the first and the last entry. This will allow us to identify patients with only a single diagnosis (ie. the first and last index are the same).

In [118]:

```
# creat new diagnosis dataframes extracting the first and last entries per patient
dx_df = pd.read_csv("../data/Cleaned/diagnosis_clean.csv", low_memory=False)
first_dx_df, first_dx_df_rows = patient_ith_entry(dx_df, index=0)
last_dx_df, last_dx_df_rows = patient_ith_entry(dx_df, index=-1)

# create new features: baseline diagnosis, final diagnosis
bl_dx = first_dx_df.DXCOMB.values
final_dx = last_dx_df.DXCOMB.values

# set final diagnosis to missing if patient only has one diagnosis
final_dx[first_dx_df_rows == last_dx_df_rows] = -1

# use first and last diagnosis to create new feature: change in diagnosis (baseline to fina
change_dx = (final_dx-1)*3 + bl_dx
change_dx[change_dx<1] = -1
```

In [119]:

```
# add features to per patient diagnosis df
pd.options.mode.chained_assignment = None  # default='warn'
first_dx_df["DX_BL"] = pd.Series(bl_dx, index=first_dx_df.index)
first_dx_df["DX_FINAL"] = pd.Series(final_dx, index=first_dx_df.index)
first_dx_df["DX_CHANGE"] = pd.Series(change_dx, index=first_dx_df.index)
first_dx_df.head()

# get the index of the per patient diagnosis dataframe and replace with the dataframe
# containing our new featuress
dx_df_idx = [i for i, path in enumerate(csv_paths) if "diagnosis" in path]
first_idx_dfs[dx_df_idx[0]] = first_dx_df
```

Combining the data

Now we can build per patient datasets by concatenating the dataframes constructed using the methods above. We'll start by combining the data aggregated by the first index method.

First entry per patient

In [120]:

```
# intialize dataframe empty placeholder
patient_first_idx_df = pd.DataFrame()

# record the data type of each column (we'll need it later)
all_dtypes = np.array([])

# iterate over dataframes
for i, df in enumerate(first_idx_dfs):

    # get columns for current df
    new_cols = df.columns

    # remove duplicate columns
    new_cols = list(set(new_cols)-set(patient_first_idx_df))

    # drop meta data and bad columns
    new_cols = list(set(new_cols)-set(["RID","Unnamed: 0","VISCODE","VISCODE2"]))

    # generate per patient dataframe from df
    tmp_df = df[new_cols]
    all_dtypes = np.append(all_dtypes,df[new_cols].dtypes.values)

    # add to patient df
    patient_first_idx_df = pd.concat((patient_first_idx_df, tmp_df), axis=1)
```

In [121]:

```
# inspect the header of the new per patient dataframe
patient_first_idx_df.head()
```

Out[121]:

RID	APGEN1	APGEN2	BCVOMIT	BCENERGY	BCHDACHE	BCURNFRQ	BCINSOMN	BCDRO
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
2	3.0	3.0	1.0	1.0	1.0	2.0	1.0	
3	3.0	4.0	1.0	1.0	1.0	1.0	2.0	
4	3.0	3.0	1.0	1.0	1.0	1.0	1.0	
5	3.0	3.0	1.0	1.0	1.0	2.0	1.0	

5 rows × 196 columns

The concatenation was successful, but in the process of concatentating, pandas automatically fills missing values in the full data set with `NaN`. The missing values can easily be converted back to `-1` with `pd.replace`, but this process of missing value filling has also converted the data types of all columns to `float64`. We will want to use the data types to distinguish categorical from continuous features. We will need to use the dtypes of the source dataframes to convert each of the columns back to the appropriate data type.

In [122]:

```
# convert missing values to -1
patient_first_idx_df.replace({np.nan:-1, -4:-1}, inplace=True)

# intialize dictionary {column: dtype,..} to set data types
dtype_dict = dict.fromkeys(patient_first_idx_df.columns.values)
for key, val in zip(patient_first_idx_df.columns.values,all_dtypes):
    dtype_dict[key] = val

# convert dtypes
patient_first_idx_df = patient_first_idx_df.astype(dtype_dict)
```

Inspect the header to check the conversion:

In [123]:

```
patient_first_idx_df.head()
```

Out[123]:

RID	APGEN1	APGEN2	BCVOMIT	BCENERGY	BCHDACHE	BCURNFRQ	BCINSOMN	BCDRO
1	-1	-1	-1	-1	-1	-1	-1	
2	3	3	1	1	1	2	1	
3	3	4	1	1	1	1	2	
4	3	3	1	1	1	1	1	
5	3	3	1	1	1	2	1	

5 rows × 196 columns

First non-missing entry per patient

Now we can do the same for the per patient dataframe assembled with the first non-missing entry method.

In [124]:

```
# intialize dataframe empty placeholder
patient_nonmissing_df = pd.DataFrame()

# record the data type of each column
all_dtypes = np.array([])

# iterate over dataframes
for i, df in enumerate(nonmissing_dfs):

    # get columns for current df
    new_cols = df.columns

    # remove duplicate columns
    new_cols = list(set(new_cols)-set(patient_nonmissing_df))

    # drop meta data and bad columns
    new_cols = list(set(new_cols)-set(["RID","Unnamed: 0","VISCODE","VISCODE2"]))

    # generate per patient dataframe from df
    tmp_df = df[new_cols]
    all_dtypes = np.append(all_dtypes,df[new_cols].dtypes.values)

    # add to patient df
    patient_nonmissing_df = pd.concat((patient_nonmissing_df, tmp_df), axis=1)
```

In [125]:

```
# convert missing values to -1
patient_nonmissing_df.replace({np.nan:-1, -4:-1}, inplace=True)

# intialize dictionary {column: dtype,..} to set data types
dtype_dict = dict.fromkeys(patient_nonmissing_df.columns.values)
for key, val in zip(patient_nonmissing_df.columns.values,all_dtypes):
    dtype_dict[key] = val

# convert dtypes
patient_nonmissing_df = patient_nonmissing_df.astype(dtype_dict)
```

In [126]:

```
# inspect the header
patient_nonmissing_df.head()
```

Out[126]:

RID	APGEN1	APGEN2	BCVOMIT	BCENERGY	BCHDACHE	BCURNFRQ	BCINSOMN	BCDRO
1	-1	-1	-1	-1	-1	-1	-1	
2	3	3	1	1	1	2	1	
3	3	4	1	1	1	1	2	
4	3	3	1	1	1	1	1	
5	3	3	1	1	1	2	1	

5 rows × 193 columns

We can confirm that there is a unique entry for each patient and a unique list of columns by comparing the unique entries of each to the full shape of the dataframe. Doing so confirms that all columns and RIDs are unique.

In [127]:

```
# confirm that there is indeed only one entry for each patient
print(patient_nonmissing_df.shape)
print(patient_nonmissing_df.index.unique().shape)
print(patient_nonmissing_df.columns.unique().shape)
```

```
(3652, 193)
(3652,)
(193,)
```

Save per patient data to file

With all the formatting and aggregation complete, it's a good idea to write the data to file as a convenient check-point we can revisit later.

In [128]:

```
patient_first_idx_df.to_csv("../data/Per_Patient/patient_firstidx_merge.csv")
patient_nonmissing_df.to_csv("../data/Per_Patient/patient_nonmissingidx_merge.csv")
```